ABSTRACT

Title of dissertation: ENHANCING PRODUCTIVITY AND PERFORMANCE PORTABILITY OF GENERAL-PURPOSE PARALLEL PROGRAMMING

Alexandros Tzannes,
Doctor of Philosophy, 2012

Dissertation directed by: Professor Rajeev Barua &
Professor Uzi Vishkin
Department of Computer Science

This work focuses on compiler and run-time techniques for improving the productivity and the performance portability of general-purpose parallel programming. More specifically, we focus on shared-memory *task-parallel languages*, where the programmer explicitly exposes parallelism in the form of short tasks that may outnumber the cores by orders of magnitude. The compiler, the run-time, and the platform (henceforth *the system*) are responsible for harnessing this unpredictable amount of parallelism, which can vary from none to excessive, towards efficient execution. The challenge arises from the aspiration to support fine-grained irregular computations and nested parallelism. This work is even more ambitious by also aspiring to lay the foundations to efficiently support ***declarative code***, where the programmer exposes *all* available parallelism, using high-level language constructs such as *parallel loops*, *reducers* or *futures*. The appeal of declarative code is twofold for general-purpose programming: it is often easier for the programmer who does not have to worry about the granularity of the exposed parallelism, and it achieves better performance portability by avoiding overfitting to a small range of platforms and inputs for which the programmer is coarsening. Furthermore, PRAM algorithms, an important class

of parallel algorithms, naturally lend themselves to declarative programming, so supporting it is a necessary condition for capitalizing on the wealth of the PRAM theory. Unfortunately, declarative codes often expose such an overwhelming number of fine-grained tasks that existing systems fail to deliver performance.

Our contributions can be partitioned into three components. First, we tackle the issue of **coarsening**, which declarative code leaves to the system. We identify two goals of coarsening and advocate tackling them separately, using static compiler transformations for one and dynamic run-time approaches for the other. Additionally, we present evidence that the current practice of burdening the programmer with coarsening either leads to codes with poor performance-portability, or to a significantly increased programming effort. This is a "show-stopper" for general-purpose programming. To compare the performance portability among approaches, we define an experimental framework and two metrics, and we demonstrate that our approaches are preferable. We close the chapter on coarsening by presenting compiler transformations that automatically coarsen some types of very fine-grained codes.

Second, we propose **Lazy Scheduling**, an innovative run-time scheduling technique that infers the platform load at run-time, using information already maintained. Based on the inferred load, Lazy Scheduling adapts the amount of available parallelism it exposes for parallel execution and, thus, saves parallelism overheads that existing approaches pay. We implement Lazy Scheduling and present experimental results on four different platforms. The results show that Lazy Scheduling is vastly superior for declarative codes and competitive, if not better, for coarsened codes. Moreover, Lazy Scheduling is also superior in terms of performance-portability, supporting our thesis that it is possible to achieve reasonable efficiency and performance portability with declarative codes.

Finally, we also implement Lazy Scheduling on XMT, an experimental many-core platform developed at the University of Maryland, which was designed to sup-

port codes derived from PRAM algorithms. On XMT, we manage to harness the existing hardware support for scheduling flat parallelism to compose it with Lazy Scheduling, which supports nested parallelism. In the resulting **hybrid scheduler**, the hardware and software work in synergy to overcome each other's weaknesses. We show the performance *composability* of the hardware and software schedulers, both in an abstract cost model and experimentally, as the hybrid always performs better than the software scheduler alone. Furthermore, the cost model is validated by using it to predict if it is preferable to execute a code sequentially, with outer parallelism, or with nested parallelism, depending on the input, the available hardware parallelism and the calling context of the parallel code.

Enhancing Productivity and Performance Portability of General-Purpose Parallel Programming.

by

Alexandros Tzannes

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2012

Advisory Committee:
Professor Rajeev Barua, Chair/Advisor
Professor Uzi Vishkin, Co-Advisor
Professor Jeff Foster
Professor Alan Sussman
Professor William Dorland

© Copyright by
Alexandros Tzannes
2012

Preface

I started my PhD in the fall of 2004. I had come to the University of Maryland at College Park to do theory, and my main interest was structural complexity. I also had an affinity to programming languages and compilers as an undergrad, and I especially enjoyed the equivalence of different automata to formal languages. In my first semester, I took approximation algorithms and randomized algorithms. Randomized algorithms, it turns out, were not my thing. I really liked approximation algorithms, but as I looked closer to the research being done in the field, I was turned off by the need to motivate theoretical research as being immediately applicable in order to apply for funding. I felt this requirement clouded the purity of theory and drove researchers away from fundamental questions which, if answered, would not necessarily provide useful fruits immediately. So, I had to reconsider what I wanted to do for my PhD.

In my second semester, I took Parallel Algorithms with Prof. Uzi Vishkin. He taught PRAM algorithms in the classroom, and we had to implement some parallel algorithms in XMTC, a small extension of C, and run the assignments on a simulator of the XMT on-chip parallel platform. In my last year of undergrad, I had taken a course on high-performance computing where we had to write code in MPI and OpenMP for a relatively simple 3D stencil computation. That was not fun. On the contrary, programming XMT was straightforward, intuitive, and fun. But the XMTC compiler was a disaster: the compiler was almost as likely to produce buggy binaries as for a programmer to write buggy code. During that time, the influential popular article by Herb Sutter [80] *"The free lunch is over: A Fundamental Turn Toward Concurrency in Software"* was calling attention to the fact that clock speeds of processors had stopped getting faster and were unlikely to pick up again. Therefore, the main path for increasing performance of code from one hardware-generation to the next would presumably be through parallelism.

That was it! I was sold on the idea of working in a "hot" field, with the

XMT architecture which I thought (and still do) was unique and very promising, combining compilers and theory, with Prof. Vishkin who I admired from undergrad for his *deterministic coin tossing* [28]. At that time, I did not yet know exactly what I wanted to do research on, but I offered Prof. Vishkin to write a decent compiler for XMTC (I had written a compiler for a large subset of C almost from scratch as an undergrad and had enjoyed it a lot, so I thought I had what it took), and Prof. Vishkin accepted the challenge of giving such a demanding project to a first year PhD student (me). He also brought on board Prof. Rajeev Barua, a compiler expert, to help with my daunting task, a very wise move. So, by the end of my first year in grad school, I had not one but two advisors Rajeev Barua and Uzi Vishkin.

During my years as a PhD student, I witnessed some parallel platforms attract significant attention but quickly loose steam, such as the IBM Cell BE, others slowly moving from domain-specific towards general-purpose, such as GPUs, some encountering difficulties to launch such as the Intel Larrabee, and others, more domain specific, trying to find a niche, such as Tilera and the Intel Single Chip Cloud. Even the now ubiquitous multicores seem to be a transitional solution, while we figure out how to adapt the hardware-software stack for parallelism.

Since I started my PhD and to this day, everyone is talking about how hard it is to write correct and efficient parallel code and how important it is to come up with practical parallel programming models, something that is still an open question [79, 75]. But instead of starting with the programming model and the algorithmic thinking and then engineering the programming languages and the platforms around them, many researchers start with a commercial product (e.g., multicores, GPUs, Cell, etc), and try to invent easy and efficient programming models for them. Undoubtedly, if someone were to succeed, the impact would be great because the hardware would already exist and the solution would be less disruptive. This approach, however, precludes quick feedback from programming models research into the development cycle of the platforms for effective codesigning the software-

hardware stack. For that reason, interest quickly migrates from one trend to the next, generating many publications but with unclear long-term contributions. For example, in the first few years of my PhD, IBM's Cell became very popular, but it has now lost most of its steam. It got replaced by the CUDA GPUs, which have had a good run, possibly because they keep evolving towards more programmable and more general-purpose parallelism. Recently, a new buzz word, *cloud computing*, is attracting a lot of interest. And the cycle starts again.

I consider myself lucky to have had advisors who stayed out of such passing fashions. It allowed me to stay focused on the bigger question of how to enable efficient and performance portable general-purpose programming. Certainly, this dissertation does not give a definitive answer to that question, but I believe it offers several steps towards that direction, including a run-time scheduler, some static compiler transformations, and a methodology for preserving performance portability of code while tuning the amount of parallelism to optimize performance. I had a great time doing research at Maryland, and I am very thankful to have the amazing opportunity to continue research in this exciting field of parallel computing at the University of Illinois at Urbana Champaign.

As a stylistic note, although dissertations are presented as the work of a single author, I am using the first-person plural as an acknowledgement of the priceless help of my advisors, research collaborators, committee members, and friends with whom I discussed my research all these years.

To my grandmother Iω.

There's nothing I can write that can begin to express my gratitude and love.

"May you always do for others
and let others do for you."

"May your heart always be joyful,
and may your song always be sung.

May you stay forever young."

Bob Dylan

Acknowledgments

I would like to extend my gratitude to all the people who made this thesis possible and due to whom my graduate experience has been one that I will cherish forever.

First and foremost, I would like to thank my advisor, Professor Rajeev Barua, who showed me his trust and patience and taught me how to do high quality research and the importance of presenting my work in a way that is accessible to a large audience. I have always had a hard time explaining my ideas, especially soon after their inception, when they are still fuzzy and not fully formed. Rajeev was always patient and helped me both with crystallizing the ideas and with presenting them in a comprehensible way.

Equally, I would like to extend my appreciation to Uzi Vishkin, my co-advisor, who taught me to wander off the beaten path and away from hypes, while staying relevant and with significant impact.

I also want to thank my dissertation committee, Prof. Jeff Foster, Prof. Alan Sussman, and Prof. William Dorland, for their time, effort, and comments, which improved this dissertation. I especially want to thank Jeff and Alan for our discussions, both on academic and other issues.

This work would have not been possible without the other members of the XMT research group who helped maintain all the software and hardware infrastructure needed for my experiments. In particular, I want to thank my close friends George C. Caragea and Fuat Keceli for sharing in the good and bad moments, as well as for their advice, support, help, and good spirits! Also the rest of the XMT team during my studies here: Xinghzhi Wen, Aydin Balkan, Beliz Saybasili, James Edwards, Darya Fillipova, Nghi Nguen Ba, Mike Detwiler, Yoni Ashar, Michael Horak, Mary Kiemb, and Tom Dubois.

I want to thank my wonderful family for their unquestioned faith in me, which was instrumental at times.

Special appreciation goes out to my close friend Manolis Georgakakis for our long discussions on all topics imaginable, for playing music together, for reluctantly sitting next to the "new guy" in high-school (that was me), and for being an inspiration.

Petros Tsamatropoulos, Nikos Prasianakis, and Dimitris Sklivagos have also blessed me with their friendship since high-school and significantly contributed to the person I am now.

I also want to thank my good friends in Maryland, Konstantinos Koutrolikos (ϑυρωρός), Kostas Spiliopoulos (πρόεδρος), Christos Papadopoulos (ψηλός), Nikos Frangiadakis (ntg), Konstantinos Bitsakos (kbits), Alex Enurah (Lex), Mike Hughes, Tiffany Dean, Vlassis Vassileiou, Thodoris Rekatsinas, Vassilis Zikas, Jimmy Hart, and Thanos Chryssis.

I am also deeply thankful to Michelle (Darth) Hugue for the privilege of having been her teaching assistant, and my professors from undergrad, Stathis Zachos and Nikos Papaspyrou, who inspired me to follow Computer Science.

I am truly indebted to Yolanda Mahnke for her support, patience, love, and understanding, especially during the months leading to the defense. I am looking forward to what comes next.

It is impossible to remember all, and I apologize to those I've inadvertently left out.

Finally, I would like to thank people who have inspired me through their timeless music, especially Ludwig Van Beethoven and Bob Dylan.

"You've been with the professors,
and they've all liked your looks.
With great lawyers you have
discussed lepers and crooks.
You've been through all of
F. Scott Fitzgerald's books.
You're very well read,
it's well known.

But something is happening here,
and you don't know what it is...

do you, Mister Jones?"

Bob Dylan

Contents

List of Tables xii

List of Figures xiii

List of Abbreviations xvi

1 Introduction 1
1.1 Contributions . . . 5

2 Background 9
2.1 XMT Background . . . 9
2.1.1 Overview of the XMT Platform . . . 10
2.1.2 From C to XMTC: Simple Extensions . . . 12
2.1.3 Performance Advantages of XMT . . . 14
2.1.4 Ease of Programming and Teaching . . . 15
2.1.5 My personal experience with the XMT Architecture . . . 16
2.2 Work Stealing Scheduling . . . 18
2.2.1 Motivation for Dynamic Scheduling . . . 19
2.2.2 Work Stealing Background . . . 21
2.2.2.1 Serializing Work Stealing (SWS): Work-First . . . 24
2.2.2.2 Help-First Work Stealing . . . 25
2.2.2.3 Eager Binary Splitting (SP & AP) . . . 25
2.2.3 Theoretical Bounds . . . 29

3 Coarsening 31
3.1 Characterizing Coarsening . . . 35
3.2 Stages of Parallelism . . . 39
3.3 Sensitivity of Performance to Coarsening . . . 41
3.3.1 Sensitivity of TBB's proposed manual coarsening . . . 42
3.3.1.1 Benchmarks . . . 43
3.3.1.2 Results . . . 46
3.3.2 Sensitivity to picking the right cut-off for QUEENS . . . 49
3.4 Evaluating the Quality of General Purpose Parallel Code: Proposed Framework . . . 52
3.4.1 Discussion . . . 56
3.4.2 Using the Framework: An Example . . . 57
3.5 Coarsening in the XMTC Compiler . . . 60
3.5.1 Cost Estimation . . . 60
3.5.2 Picking a Grain-Size . . . 61
3.5.3 Serializing Spawn Statements (Parallel Loops) . . . 62
3.6 Conclusion and Future Directions . . . 62

4 Lazy Scheduling 64
4.1 The two Insights of Lazy Scheduling 64
4.2 Lazy Binary Splitting (Depth-First Lazy Work Stealing) 66
4.3 Analytical Comparison of Lazy Work Stealing with existing Work Stealing schedulers: A First approach 68
4.3.1 Deque Checks 73
4.3.2 Role of the Profitable Parallelism Threshold (*ppt*) 73
4.4 Experimental Evaluation of Depth-First Lazy Work Stealing (LBS) on XMT 74
4.4.1 Scalability and Speedups 82
4.5 Scalability Issues of Depth-First Lazy Work Stealing 83
4.6 Lazy Scheduling for Declarative Code 89
4.6.1 Breadth-First Lazy Scheduling (BF-LS) 89
4.6.2 DF-LS with a threshold of 2 (DF2-LS) 93
4.7 Experimental Evaluation of Lazy Work Stealing on Multicores 95
4.7.1 Scaling of Lazy Scheduling on Multicores 95
4.7.2 Counting Thefts 97
4.7.3 Evaluation on a set of benchmarks 98
4.7.4 Software Optimality of Declarative Code 107
4.7.5 Software Optimality of Code with Amortizing Coarsening . . . 109
4.7.6 Worst-Case Software Optimality of Lazy Scheduling 111
4.8 Experimental Evaluation of Scalable Lazy Work Stealing on XMT . . 113
4.8.1 Scaling of Lazy Scheduling on XMT 114
4.8.2 Counting Thefts 116
4.8.3 Evaluation on a set of Benchmarks 116
4.9 Related Work 117
4.9.1 Schedulers without Parallel Loop Support 118
4.9.2 Schedulers with Parallel Loop Support 120
4.9.3 Parallelism Throttling 123
4.10 Analytical Comparison with other Work Stealers: A Second Approach 124
4.10.1 Space Bounds 124
4.10.2 Time Bounds 125
4.10.3 Adversarial Scenarios for AP and Lazy Scheduling 126
4.11 The Inception of Lazy Scheduling: an Interesting Anecdote 128

5 The XMTC Compiler 130
5.1 Overview 130
5.2 The XMTC Memory Model 134
5.3 Compiling XMTC Parallel Code with a Serial Compiler 137
5.3.1 Outlining 138
5.3.2 Register Broadcasting 138
5.3.3 Assembly Code Layout Correction 140
5.3.4 Why illegal dataflow is not an issue for thread libraries 141
5.4 XMT-Specific Optimizations 142
5.4.1 Latency tolerating mechanisms. 142

5.4.1.1 Burst Prefetching 143
5.5 Compiling a flat XMTC spawn statement 149
5.6 Nested Parallelism Support 152
5.6.1 Function Cloning 152
5.6.2 Function Call Support in Parallel Code: Stack Allocation . . . 153
5.6.3 Function Insertion 162
5.6.4 Dead Function Elimination 162
5.6.5 Outlining Optimizations 163
5.6.6 Compiling a nested spawn statement 164
5.6.7 Outer Spawn Compilation for Nesting 169

6 Model of Scheduling Costs and Architectural Support 173
6.1 Background 176
6.1.1 Algorithmic Models 176
6.1.2 Background for Modeled Schedulers 179
6.2 A Work-Depth Model for XMT's Hardware and Work Stealing Software Schedulers 181
6.3 Evaluation of the Scheduling Models 183
6.3.1 Measuring Q, a, b 184
6.3.2 Methodology for measuring Q, a, b 185
6.3.3 Orthogonality of Hardware and Software Scheduling on XMT 188
6.3.4 Using the model to predict the preferable scheduling option . 192
6.4 Related Work 193
6.5 Conclusions and Future Directions 194

7 Conclusion and Future Directions 196
7.1 Future Work 197

Bibliography 199

List of Tables

3.1 Summary of XMTC Benchmarks . . . 45
3.2 Benchmarks, Datasets, and Grain-Sizes. . . . 46
3.3 Worst-Case Software Optimality with constant cut-offs. . . . 58
3.4 Worst-Case Software Optimality with cut-off functions. . . . 59
3.5 Example of Functional Unit Costs used for Task Cost Estimation. . . 61

4.1 Transaction and Synchronization Costs . . . 70
4.2 Summary of XMTC Benchmarks . . . 75
4.3 Benchmarks, Datasets, and Grain-Sizes. . . . 75
4.4 Standard Deviation(%) for Recursively Nested Benchmarks. . . . 76
4.5 Summary of Compared Configurations . . . 76
4.6 Speedups of LBS+ vs. Parallel Program on 1 TCU . . . 82
4.7 Speedups of LBS+ vs. Serial Program on MTCU . . . 83
4.8 Platform Descriptions . . . 86
4.9 Standard Deviation(%) for i7 (Figures 4.7 and 4.12.) . . . 87
4.10 Standard Deviation(%) for Xeon (Figures 4.8 and 4.11.) . . . 87
4.11 Standard Deviation(%) for T2 (Figures 4.9 and 4.10.) . . . 87
4.12 Comparison of schedulers. . . . 94
4.13 Number of thefts (Average over 10 runs) . . . 98
4.14 Benchmark Summary . . . 99
4.15 Standard Deviation(%) for i7 (Figure 4.13.) . . . 102
4.16 Standard Deviation(%) for Xeon with 6 workers (Figure 4.14.) . . . 103
4.17 Standard Deviation(%) for T2 with 8 workers (Figure 4.15.) . . . 103
4.18 Standard Deviation(%) for Xeon with 24 workers (Figure 4.16.) . . . 105
4.19 Standard Deviation(%) for T2 with 64 workers (Figure 4.17.) . . . 105
4.20 Standard Deviation(%) for T2 (Figure 4.18.) . . . 107
4.21 Software Optimality (%) of Declarative Code on i7 . . . 108
4.22 Software Optimality (%) of Declarative Code on Xeon . . . 108
4.23 Software Optimality (%) of Declarative Code on T2 . . . 108
4.24 Amortizing Cut-Off Depths for QUEENS and TSP . . . 110
4.25 Software Optimality (%) of Amortized Code on i7 . . . 111
4.26 Software Optimality (%) of Amortized Code on Xeon . . . 111
4.27 Software Optimality (%) of Amortized Code on T2 . . . 111
4.28 Worst-Case Software Optimality for BF-LS. . . . 112
4.29 Standard Deviation(%) for XMT with 64 workers (Figure 4.20.) . . . 118
4.30 Standard Deviation(%) for XMT- with 64 workers (Figure 4.21.) . . . 118
4.31 Space and Time Bounds for Generic Parallel Loop . . . 125

6.1 WD equations for a parallel-loop with N tasks . . . 182
6.2 Work and depth equations for hardware, software, and hybrid scheduling. . . . 184
6.3 Values in cycles for different values of N (with $N = M = L$). . . . 185
6.4 Values are in cycles for different values of N (with $N = M = L$). . . . 186

List of Figures

2.1 Overview of the XMT architecture. 10
2.2 (a) XMT Execution Modes. (b) XMTC Code Example. 12
2.3 A Case for Dynamic Scheduling: Load Imbalance 19
2.4 Quicksort in XMTC: Example of Recursively Nested Parallelism. . . . 20
2.5 Processing a Task Descriptor with Simple-Partitioner. 26

3.1 Amortizing Overheads vs. Pruning Parallelism 37
3.2 Performance Sensitivity of TBB's Manual Tuning 47
3.3 Sensitivity of Performance when varying the number of workers . . . 49
3.4 Sensitivity of Performance when varying the size of the input 51

4.1 Processing a Task Descriptor with Lazy Binary Splitting (LBS) . . . 66
4.2 Comparing LBS to AP_{xmt} and $AP_{default}$ 77
4.3 Comparing LBS to $SP_{tr/ex}$ 78
4.4 Comparing LBS to $SP_{ex/ex}$ 79
4.5 Comparing LBS and LBS_1 to SP_1 and SWS 80
4.6 Comparing LBS+ to SI 82
4.7 Scaling of schedulers on i7 (Queens) 86
4.8 Scaling of schedulers on Xeon (Queens) 88
4.9 Scaling of schedulers on T2 (Queens) 88
4.10 Scaling of schedulers on T2 (Queens) 96
4.11 Scaling of schedulers on Xeon (Queens) 96
4.12 Scaling of schedulers on i7 (Queens) 97
4.13 Benchmarks on the i7 using all 8 Workers 101
4.14 Benchmarks on the Xeon using only 6 Workers 101
4.15 Benchmarks on the T2 using only 8 Workers 102
4.16 Benchmarks on the Xeon using all 24 Workers 104
4.17 Benchmarks on the T2 using all 64 Workers 104
4.18 Scaling of schedulers on T2 (Fib(36)) 106
4.19 Scalability of LBS & BF-LS on XMT, XMT- and XMT--. 114
4.20 Benchmarks on XMT using all 64 TCUs. 116
4.21 Benchmarks on XMT- using all 64 TCUs. 117

5.1 Compiler Passes. 131
5.2 Two tasks with no order-enforcing operations or guarantees. 134
5.3 Enforcing partial order in the XMT memory model. 136
5.4 Simple example of outlining. 139
5.5 Example of assembly basic-block layout issue. 140
5.6 Simple Burst Prefetching Example. 144
5.7 Complete Burst Prefetching Example. 145
5.8 Compiling a flat spawn statement 149
5.9 Identifying Outer and Nested Spawns 152
5.10 Function Cloning 153
5.11 Spawn Scope Example 154

5.12 Stacklet 155
5.13 Compiling a flat spawn with cactus stack support 161
5.14 Hoisting Low & High Expressions 164
5.15 Common Preamble of all frame structures. 167
5.16 The task descriptor structure 167
5.17 Outer Spawn Conversion 169
5.18 Compiling an outer spawn to support nesting 170

6.1 A common case for low-degree parallelism: BFS 175
6.2 Increasingly abstract algorithmic models 177
6.3 Parallel Matrix Multiplication 184
6.4 Hybrid vs. hardware: insufficient outer parallelism. 188
6.5 Hybrid outperforms hardware: load imbalance. 189
6.6 Break-even point with sequential code. 190
6.7 Hardware and hybrid scheduling are complementary. 191

List of Algorithms

3.1 Fully Parallel Matrix Multiplication 32
3.2 Matrix Multiplication: Parametrically Coarsened 33
3.3 Queens pseudocode. $depth \in [1, N]$ 36
4.1 Queens declarative pseudocode. $depth \in [1, N]$ 85
4.2 BF-LS Scheduling of a TD representing a parallel loop 91
4.3 Generic Parallel Loop 124
5.1 Burst Prefetching Algorithm 146
5.2 Burst Prefetching Pseudocode 148
5.3 Function Prologue Expansion for supporting Stacklets 157
5.4 Stacklet Deallocation 158
5.5 Function Prologue Expansion for Outer-Spawn function 160
5.6 Outlining Inner Spawn: original code 165
5.7 Outlining Inner Spawn: resulting code 166
5.8 Scheduling loop for work stealing 171

List of Abbreviations

AfP	Affinity-Partitioner
AP	Auto-Partitioner
BF-LS	Breadth-First Lazy Scheduling
BFS	Breadth-First Search
CONV	Convolution
DAG	Directed Acyclic Graph
DF-LS	Depth-First Lazy Scheduling (a.k.a. LBS)
DF2-LS	Depth-First Lazy Scheduling with a deque threshold of 2
FIFO	First-In First-Out
FW	Floyd-Warshall all-pairs shortest path algorithm
IOS	Independence of Order Semantics
IWD	Informal Work Depth
LBS	Lazy Binary Splitting
MM	Matrix Multiplication of dense matrices
OPT	(scheduling) Overhead Per Task
OS	Operating System
ppt	profitable parallelism threshold
PRAM	Parallel Random Access Machine
QSort	Quicksort algorithm
RAM	Random Access Memory
SP	Simple-Partitioner
SpMV	Sparse Matrix by Vector multiplication
sst	stop-splitting threshold
SWS	Serializing Work Stealing
TBB	Threading Building Blocks
TD	Task Descriptor
TSP	Traveling Salesman Problem
WD	Work Depth
XMT	eXplicit Multi-Threading architecture
XMTC	A parallel extension of C for programming XMT

Chapter 1

Introduction

Technological reasons have led processor manufacturers to abandon scaling the clock frequency and to turn to parallelism to improve performance. This disruptive change has and will have a fundamental impact on how programmers write code that takes advantage of the available parallel hardware to build complex applications. The central goal of this dissertation is to provide some guidelines and some solutions towards bringing parallel programming to the mainstream and making it successful.

This work focuses on compiler and run-time techniques for explicitly parallel codes because, after decades of research in automatic parallelization, it seems unlikely that compilers will be able to shoulder in the near future the responsibility of parallelizing all types of sequential code efficiently. While compilers can deliver good results on regular affine codes, irregular applications pose a great challenge, especially because programmers often use data-structures (e.g., FIFO queues) in their serial code that hide the inherent code parallelism from the compiler.

Explicit parallel programming, where the programmer has to identify what can be executed in parallel, seems like a necessary alternative to automatic parallelization. Nevertheless, there are many different types and flavors of parallel programming, depending on the application domain (e.g., data and task parallelism, dataflow, streaming), the scale or hardware architecture targeted (e.g., shared versus distributed memory), and many other parameters. Literally hundreds of parallel programming languages have been proposed in an attempt to make parallel programming easier, but few have been successful in deployment.

Different factors complicate parallel programming in different paradigms (e.g., deadlocks, races, partitioning the data, orchestrating communication/synchronization, etc.), but this dissertation focuses on shared-memory *task-parallel languages*,

where the programmer exposes parallelism in the form of short tasks that may outnumber the cores by orders of magnitude. It is the responsibility of the run-time scheduler to efficiently distribute and execute this abundance of tasks onto the available hardware (or threads). Cilk [38] was probably the language that popularized this approach, and industry is currently in the process of adopting it. Intel has developed Threading Building Blocks (TBB) [76], a library that enables programmers to write in this style, as well as CilkPlus, Intel's most recent effort to commercialize Cilk++ after acquiring Cilk Arts. Microsoft's Task Parallel Library (TPL) [61] is a library with similar goals as TBB. Finally, the Java Fork-Join model [59] also implements task parallelism.

One benefit of task-parallel languages is to provide programmers with high-level parallel constructs, such as parallel do-all loops, sum-like reductions, and parallel function calls or futures, which have implicit synchronization semantics and greatly simplify coding; furthermore, these parallel constructs can be freely nested, allowing to create parallel tasks from within parallel tasks. This support of *nested parallelism* is important for three reasons: *(1)* it ***enables modularity*** in parallel programming by allowing to call a function that creates parallelism from sequential and parallel contexts alike; *(2)&(3)* when more parallelism becomes available as the computation progresses and the first set of parallel tasks to become available does not contain enough parallelism or is composed of load imbalanced tasks, nested parallelism dissects these outer parallel tasks and ***increases the available parallelism*** and ***improves load balance***. Both result in performance improvements.

One of the issues programmers face is ***task granularity***, a double edged sword: if they expose tasks that are too fine-grained, the scheduler will not be able to execute them efficiently, but if they do not expose enough parallelism, performance will be elusive once again because of insufficient parallelism or because of load imbalance, when some tasks finish earlier than others and leave processors idling. So, it falls on the programmer's shoulders to adequately coarsen task paral-

lelism to achieve good performance.

Unfortunately, performance is sensitive to coarsening choices, as we will show in Chapter 3. Coarsening should take into account *(1)* the amount of available task parallelism, which often depends on the size or shape of the input data D, *(2)* the number of available processors and other platform characteristics P, and *(3)* the context C in which a parallel code is invoked; if invoked from a sequential context (i.e., one execution thread), performance will benefit from more parallelism (less coarsening) than if invoked from a parallel context. Because the performance is sensitive to parameters (D, P, C) that are not necessarily known at compile-time, the programmer must either coarsen in a parametric way, taking into account the run-time values of these parameters, or coarsen for a relatively restricted subdomain of D, P, and C. Although such *subdomain coarsening* is perfectly legitimate for high-performance parallel computing where expert programmers focus and optimize applications of interest for a specific target platform, for *general-purpose* parallel programming it is not viable.

General-purpose parallel programming, like its sequential counterpart, treats *ease of programming* and *performance portability* (also called performance robustness) as first-order considerations, and it strives to achieve the best possible performance within these constraints. Ease of programming is hard to define rigorously, but one definition could be to require minimal detail inclusion in the abstract machine model the programmer uses while programming. For example, the RAM (Random Access Machine) model used for sequential programming assumes unit time access to a memory of unbounded size. Performance portability requires a single unmodified code to perform well on a large set of different platforms. Manual coarsening relates to these issues because parametric coarsening (in D, P, C) sacrifices ease of programming, whereas subdomain coarsening harms performance portability.

As a solution, it has been proposed to allow the programmer to express *all*

available parallelism, no matter how fine-grained, and expect the compiler, run-time system, and hardware to execute this abundance of parallelism efficiently. We will refer to code that exposes all parallelism using high-level parallel constructs as *declarative*. It is true that using such constructs often imposes somewhat conservative synchronization patterns and does not truly expose all possible parallelism, but exposing all parallelism in an unstructured way would be harder for the programmer, and the unstructured synchronization patterns would probably complicate the scheduling code and increase its overheads, like in the case of full-fledged threads (e.g., Pthreads synchronization costs are much higher than the often implicit synchronization of tasks). For those reasons, parallelism needs to be expressed through high-level parallel language constructs. In fact, we believe there is a parallel between the argument for structured parallelism and that for structured sequential code made in favor of dropping the use of `goto` statements [32].

The thesis of this dissertation is the following:

> *It is possible to efficiently support declarative code in the context of general-purpose parallel programming.*

The results of this dissertation will help to restore the rightfully shaken belief [74] that support of declarative code can be made efficient. Efficiently supporting declarative code *enhances productivity* by not requiring tedious manual coarsening of parallelism and *improves performance-portability* by enlisting advanced compiler and run-time techniques to maintain the flexibility of declarative code.

More specifically, this dissertation advocates to partition the challenge of coarsening presented by declarative code into two disjoint components and attack each separately. We propose *Lazy Scheduling*, a technique that effectively coarsens parallelism at run-time based on load conditions, in order to achieve one of the two

components of coarsening. We convert the industry-standard work stealing scheduling algorithm to make it lazy, and we show the significant advantages of Lazy Work Stealing on declarative code on a number of platforms and benchmarks. Furthermore, we provide compiler transformations that partially resolve the other component of coarsening, and we show that, in the hypothetical but realistic situation where compilers and other technologies can completely automate that step, declarative code scheduled with Lazy Scheduling can achieve very high efficiencies.

1.1 Contributions

This section lists the technical contributions this dissertation provides towards improving the efficiency of declarative task-parallel codes.

Lazy Scheduling. The main technical contribution of this dissertation is *Lazy Scheduling*, a novel scheduling algorithm that effectively adapts the granularity of parallelism to run-time load conditions. The first insight of Lazy Scheduling is that it avoids exposing parallelism on the shared work-pool when workers are busy. This greatly reduces scheduling overheads and enables efficient support of much finer granularity of parallelism than previously possible. The second insight of Lazy Scheduling is a lightweight and scalable heuristic for inferring the system-load. We originally implemented Lazy Scheduling in the XMTC language for the XMT architecture, and later, we also implemented it in Intel's TBB library. We compare Lazy Scheduling to the state of the art schedulers using a set of benchmarks on different platforms, and demonstrate important performance improvements on fine-grained (e.g., declarative) codes, without harming performance portability. This work is presented in Chapter 4.

Characterizing Coarsening. We start by characterizing coarsening of parallelism based on its two goals: *amortizing* the scheduling overhead per task (OPT) and *pruning* parallelism. We proceed to argue why these two goals are separate in the presence of nested parallelism and why they should be tackled by different

methods. Roughly speaking, the amount of pruning needed depends on the input D, the platform P, and the context C, which are often unknown statically[1], so pruning should be done dynamically when this information becomes available to avoid hurting performance portability. On the other hand, amortizing the OPT by definition has to happen before tasks reach the scheduler. To that effect, static coarsening, automatic or manual, and just-in-time compilation can provide solutions. We have found that the OPT has only small variations across the platforms we have explored, which makes static coarsening a good solution that does not hurt performance portability.

While these goals are probably clear or at least implicit in the minds of the experts of the field, we have not found written documentation characterizing the difference between amortizing the scheduling overhead per task and pruning parallelism, let alone seen arguments that the two goals should be achieved independently. On the contrary, existing approaches try to achieve both goals by manual coarsening, which often results in code that is not performance portable. Chapter 3 discusses these issues, as well as the following contributions of this work relating to coarsening.

Coarsening Sensitivity & Performance Portability Metrics. To illustrate the pitfalls of manual coarsening and to motivate the importance of declarative code, we show that subdomain coarsening (i.e., non parametric coarsening) harms performance portability. On the other hand, we show that parametric coarsening is trickier and harms ease of programming. We show that even applications that seem simple to coarsen manually are sensitive to coarsening. To that end, we define the *worst-case software optimality* and the *average-case software optimality* of a code, and we use these metrics to evaluate the performance portability achieved by different manual coarsenings (Chapter 3).

Automatic Static Coarsening. Lazy Scheduling is married to some compiler optimizations that attempt to detect extremely fine-grained tasks and coarsen them

[1]Note that modular programming hides the context in which a code is executed.

either statically or at run-time, when more information is available (Chapter 3). For example, the number of tasks of a parallel loop may not be known statically, but it is known right before starting to execute the loop. These compiler-coarsening optimizations are parametric and, therefore, do not harm performance portability. However, they do not detect all types of fine-grained codes, most notably recursively nested parallelism found in algorithms such as Quicksort or solving the traveling salesman problem (TSP). Nevertheless, Lazy Scheduling manages to achieve reasonable efficiency even on those algorithms without static coarsening. When we applied manual coarsening to those algorithms to amortize the OPT, we found that Lazy Scheduling was able to prune the remaining excess parallelism and to achieve very high efficiency without compromising the performance portability of the code (Chapter 4).

The XMTC compiler. Chapter 5 presents the XMTC compiler, which uses GCC and CIL as building blocks. The contributions are the following: *(1)* we present the lessons learned on how to modify a compiler for a sequential language to target a parallel language, without completely overhauling the compiler internals; *(2)* we present some novel compiler transformations and optimizations that are specific to XMT; *(3)* we brought the stability of the XMTC compiler to a high enough level, enabling the teaching of parallel algorithms to all levels of education, from high-school to doctoral, and enabling the publication of several research papers [33, 77, 19, 20, 82, 73, 54, 24, 51, 53].

Modeling the Runtime. Finally, as we aspire to completely automate coarsening in the future, we propose a parametric model for costs of the runtime in Chapter 6. This model estimates when to execute a set of tasks sequentially rather than in parallel and how much to coarsen them in the latter case. Auto-tuning can be used to determine the constant parameters of the model on a given platform at the time of the compiler installation. The compiler can then use the model and the computed values of its parameters to make more accurate coarsening decisions. This part of

the dissertation represents exploratory work, as further validation and refinements to the model are necessary. Furthermore, we use the model to show an orthogonal relationship between the hardware scheduling offered by the XMT platform and the software schedulers discussed in this work. This means that the added hardware does not have harmful effects on performance under any circumstance and that on XMT the combination of hardware plus software scheduling is always preferable to software scheduling alone. We present experimental evidence supporting this and show instances where the hardware improves performance.

Chapter 2

Background

This chapter is divided into two sections: the first gives the necessary background on the the eXplicit Multi-Threading (XMT) on-chip general-purpose computer architecture and its programming language, XMTC; the second section introduces the popular and widely used work stealing scheduling algorithm and its variations, since it inspired lazy scheduling, presented in Chapter 4. The advantages of lazy scheduling over work stealing are explored both on XMT and on commodity multicores.

2.1 XMT Background

As mentioned in the introduction, the central goal of this dissertation is to improve compiler and run-time support for declarative parallel code. While this work constitutes a step forward in that direction, it is important to recognize that hardware support is also a necessary component for efficiently supporting all types of declarative code. The XMT architecture developed at the University of Maryland provides an excellent platform for supporting declarative parallel code, as it was designed to efficiently support the fine-grained and irregular parallelism that comes from PRAM algorithms. Moreover, XMT allows to achieve speedups with significantly less program parallelism than commodity platforms, and by combining XMT with the contributions of this dissertation that focus on executing efficiently excessive amounts of parallelism, we propose a solution that is optimal for both extremes of too much and too little program parallelism. In this section, we present some background on XMT and its programming language, XMTC, which is a simple parallel extension of C.

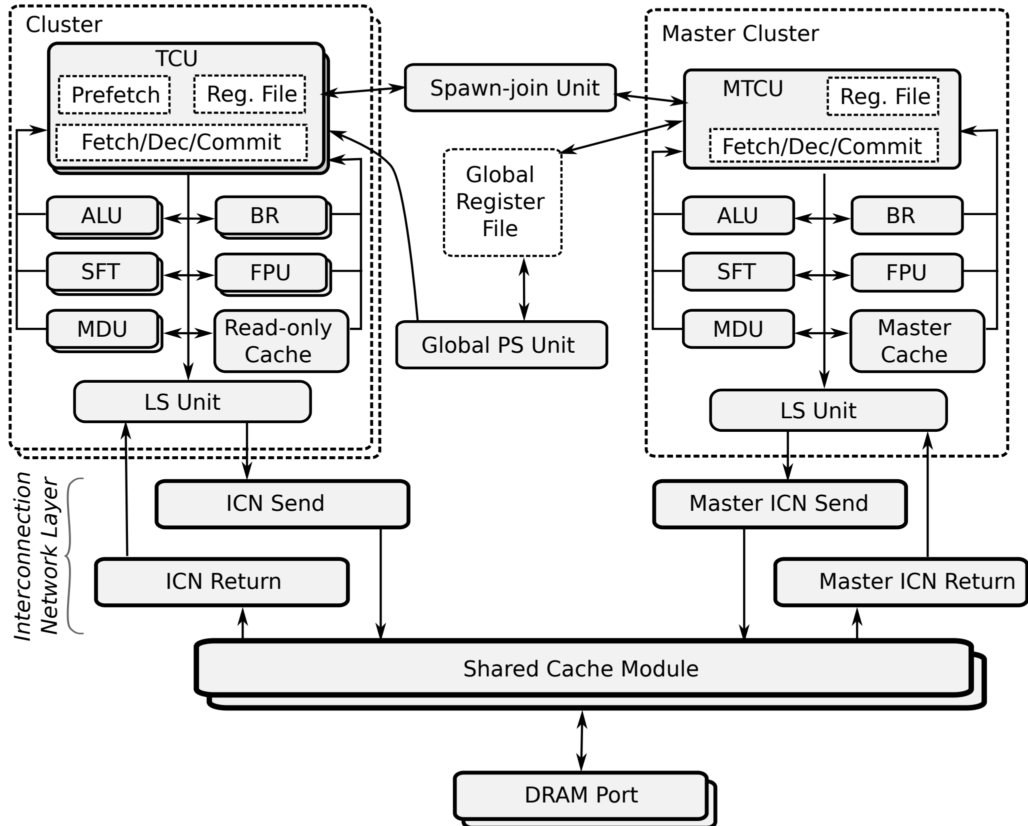


Figure 2.1: Overview of the XMT architecture.

2.1.1 Overview of the XMT Platform

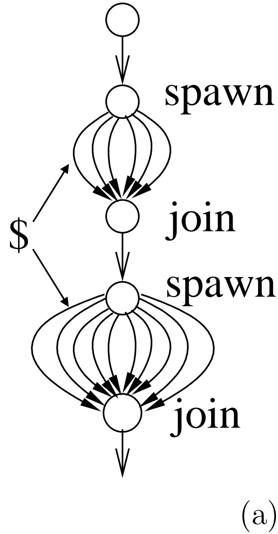
The primary goal of the XMT architecture [88, 68, 69] has been to improve single-task performance through parallelism. XMT was designed from the ground up to capitalize on the huge on-chip resources becoming available in order to support the algorithmic body of knowledge known as Parallel Random Access Model (PRAM) algorithmics [50, 55] and the latent, though not widespread, familiarity with it. A 64-core FPGA prototype was reported and evaluated by Wen et al. [91, 92].

The XMT architecture, depicted in Figure 2.1, includes a multitude of lightweight cores, called Thread Control Units (TCUs), and a sequential core with its own cache, the Master TCU. TCUs are grouped into clusters, which are connected by a high-throughput interconnection network to the first level of cache (L1), using, for example, a mesh-of-trees topology [10, 12]. The L1 cache is shared among

TCUs and partitioned into mutually-exclusive cache modules, sharing several off-chip DRAM memory channels. The load-store (LS) unit applies hashing on each memory address to avoid hotspots. Cache modules handle concurrent requests, which are buffered and reordered to achieve better DRAM bandwidth utilization. Within a cluster, a read-only cache is used to store shared variables with constant values. Each TCU includes a lightweight arithmetic logic unit (ALU), a shift unit (SFT), and a branch (BR) unit, but the more expensive multiply/divide (MDU) and floating-point units (FPU) are shared among TCUs in a cluster. Each TCU also features prefetch buffers, which are utilized by compiler optimizations to overlap memory operations and hide latencies.

XMT allows concurrent instantiation of as many tasks as there are TCUs. Tasks are efficiently started and distributed thanks to a dedicated data and instruction data broadcast bus as well as with the help of a custom hardware prefix-sum operation for fast dynamic allocation of tasks to TCUs. XMT also provides hardware support to perform a barrier-type operation over all tasks running in parallel for efficiently switching back to sequential execution. The high-bandwidth interconnection network (ICN) and the low-overhead creation of many tasks facilitate the efficient support of fine-grained, irregular, and low-degree parallelism. XMT also efficiently supports applications with regular, coarse-grained, and abundant parallelism, but without necessarily providing such a clear advantage over other architectures.

XMT operates in one of two *execution modes*: sequential or parallel. Programs start in sequential mode and alternate between parallel and sequential modes. Sequential portions of the code are executed in sequential mode by the powerful Master TCU, whereas parallel portions of the code are executed in parallel mode by the plethora of lightweight TCUs. Figure 2.2(a) illustrates the transitions between execution modes induced by *spawn statements*, which introduce parallelism in the XMTC programming language.



(a)

```
int A[N],B[N],base=0;
spawn(0,N-1) {
    int inc=1;
    if (A[$]!=0) {
        ps(inc,base);
        B[inc]=A[$];
    }
} // implicit join
```

(b)

Figure 2.2: (a) XMT Execution Modes. (b) XMTC Code Example.

2.1.2 From C to XMTC: Simple Extensions

XMTC, the programming language of XMT, is a modest extension of C with three new keywords that relate to parallel execution (`spawn, ps, psm`). The example in Figure 2.2(b) illustrates the use of the first two keywords with a simple but interesting example. Like Cilk or OpenMP programs but unlike UPC or MPI programs, XMTC programs start in sequential mode using the MasterTCU. When parallelism is encountered, the execution switches to parallel mode. The execution may switch between execution modes multiple times and finally return to sequential mode to terminate. This model of execution is considered by some to follow the SPMD model (Single Program Multiple Data). Others consider UPC or MPI codes to be SPMD, where all available cores simultaneously start executing and there is no sequential execution mode. Sequential sections can be emulated by having one core execute the sequential portion, while the others wait (e.g., by spin-waiting at a barrier).

The *spawn statement* introduces parallelism in XMTC. It is a type of parallel loop, whose iterations *can* be executed in parallel. It takes two arguments `low`, and `high`, and a block of code, the *spawn block*. Conceptually, the block is concurrently

executed by $(high - low + 1)$ *tasks*. The unique identifier of each task can be accessed using the dollar sign ($) and takes integer values within the range $low \leq \$ \leq high$. Variables declared in the spawn block are private to each task, whereas variables declared in the enclosing scope are shared by the spawn tasks. All tasks must complete before the execution proceeds beyond the spawn statement. In other words, a spawn statement introduces an implicit synchronization point at the end of its spawn block. The number of tasks created by a spawn statement is independent from the number of TCUs in the XMT system and often significantly exceeds that number.

XMTC also provides access to XMT's powerful hardware prefix-sum (`ps`) primitive, similar in function to the NYU Ultracomputer atomic Fetch-and-Add [42]. It enables constant time, low-overhead coordination between tasks, a key requirement for implementing efficient fine-grained parallelism. The prefix sum takes two arguments, a *base* and an *increment*, and performs the following two actions atomically: *(1)* it adds the increment to the base, and *(2)* it returns the original value of the base. Although the `ps` operation is efficient, it can only be performed over a limited number of hardware global registers and only with increment values of 0 (read) and 1 (increment by 1). For that reason, XMTC also provides a prefix-sum to memory variant (`psm`), that does not have these limitations: the *base* can be any memory location, and the value of the *increment* can be any signed (32 bit) integer. The `psm` operations are more expensive than `ps`, however, as they require a round trip to memory, and multiple `psm` operations that arrive simultaneously at the same cache module will be queued. By contrast, the prefix-sum hardware will combine multiple concurrent `ps` operations and service all of them in constant time.

The XMTC example code in Figure 2.2(b) performs array compaction: the non-zero elements of array A are copied into array B; the order is not necessarily preserved. The dollar sign ($) refers to the unique task identifier. After the execution of the prefix-sum statement `ps(inc,base)`, the `base` variable is increased by `inc`

and the `inc` variable gets the original value of `base`, as an **atomic** operation. Thus, the `ps` operation is used here to acquire the next available index in the target array B, where the non-zero elements of array A are then stored.

An XMTC program derived from a PRAM algorithm following the XMT workflow [87] permits each task to progress at its own speed, without ever having to *busy-wait* for other tasks. This property is called *independence of order semantics* (IOS). In the array compaction example above, this is achieved by having each task acquire the next available index in B using the prefix-sum operation. Since `ps` takes constant time and does not incur queuing overheads, tasks can proceed at their own speed without waiting for other tasks.

More details on XMTC can be found in the XMT Toolchain Manual [23]. Note that the single-spawn `sspawn` statement described therein is a remnant of the time before we added convenient and efficient support for nested parallelism in XMTC. Its use is strongly discouraged because it is error-prone and, in most cases, not efficient. Moreover, combined use of single-spawn and nested spawn statements in a parallel section is illegal. For these reasons, we will not further discuss the single-spawn in this dissertation.

2.1.3 Performance Advantages of XMT

A cycle-accurate 64-core FPGA hardware prototype [91, 92] was shown to outperform an Intel Core 2 Duo processor [22], despite the fact that the Intel processor uses more silicon resources. The XMT simulator was used to compare a 1024-TCU XMT chip to a silicon-area equivalent GPU, namely the nVidia GTX280. Simulations revealed that, in addition to being easier to program than the GPU, XMT has the potential of coming ahead in performance [19] within the same thermal constraints as the GPU [53]. Another comparison with GPUs can be found in [33]. A comparison of FFT (the Fast Fourier Transform) on XMT and on multi-cores showed that XMT can both get better speedups and achieve them with less appli-

cation parallelism [77]. The XMTC compiler was an essential component in these experiments and in more publications on XMT.

2.1.4 Ease of Programming and Teaching

Ease of programing and productivity are central objectives for XMT and XMTC. Given that ease of teaching is a necessary condition for ease of programming, demonstrating the teachability of XMTC has been a focal point of the XMT group efforts. Since 2007, more than 100 students in high-schools have been taught to program XMT, including two magnet programs: Montgomery Blair High School, Silver Spring, MD, and Thomas Jefferson High School for Science and Technology, Alexandria, VA. In fact, at Thomas Jefferson, Torbert[82] has incorporated XMT into their curriculum and advocates using it broadly in Computer Science education. More specifically, Torbert reports that, when compared to MPI, with XMT *"it was no longer the case that everyone in the lab was chasing the same canonical solution, but, instead, students were actually inventing different methods for solving these problems."*

In a semester-long study supported through the DARPA HPCS program, the development time of XMTC was shown to be about half that of MPI, under circumstances favoring MPI [48]. Some circumstantial evidence in [19] and [82] also suggests that XMTC is easier than CUDA.

In a joint teaching experiment between the University of Illinois and the University of Maryland comparing programming in OpenMP and in XMTC [73], none of the 42 students achieved speedups greater than one using OpenMP programming for the simple irregular problem of breadth-first search (BFS) using an 8-processor SMP (Symmetric Multi-Processor), but they reached speedups of 7x to 25x on the 64-TCU XMT FPGA. Moreover, the PRAM/XMT part of the joint course was able to convey algorithms for more advanced problems than the other parts.

That is the other advantage of XMT in terms of ease-of-programming, namely

that it is designed to support PRAM algorithms and that it provides a *programmer's workflow* for deriving efficient programs from PRAM algorithms, and reasoning about their execution time [87] and correctness. This workflow guides programmers in converting a PRAM algorithm to an XMTC program, which makes it easier to avoid the pitfalls of parallel programming. Moreover, this workflow allows changing the program incrementally to optimize performance, without having to redesign the basic algorithm. This is in direct contrast with the Culler-Singh [31] 4-step programming-for-locality recipe: decomposition, assignment, orchestration, and mapping, which is often difficult and may require a complete redesign of the program if, say, the decomposition step is conceptually altered.

2.1.5 My personal experience with the XMT Architecture

Because "compilers and run-time systems for parallel programming" is a man-made field that is constantly changing, the choice of assumptions (or model) is paramount: they can be enablers or disablers for the significance and robustness of the work. Since a compiler translates input code into output code, compiler research must chose two sets of assumptions, or models: the parallel languages (the input) and the parallel platforms (the output). Even during the relatively short duration of this work, we have seen industry endorsed platforms such as the Intel Itanium and Larrabee, the IBM Cell, and various GPU architectures that either changed dramatically, lost steam, or disappeared altogether. The biggest constraints on writing a dissertation in this field is that the choice of models needs to take place relatively early. For this reason, I thought that it might be appropriate to review my own original skepticism regarding the XMT architecture, and how it dissipated as I dug deeper and gained experience and insight through study and experimentation.

There were three design choices of XMT that raised concerns for me in my early stages of contributions to the XMT project, but which I now understand as being sound. First, the sharing of resources within clusters of TCUs, such as the

multiply-divide unit, raised questions as to whether XMT could perform on par with existing architectures that are efficient for parallel numerical computations, such as matrix multiplication. The shared functional units are pipelined, which means they can complete one operation per cycle and the overhead of sharing them is a queuing delay equal to the number of TCUs sharing it, in the worse case. As long as the number of TCUs sharing a functional unit is not larger than the average latency of the unit, the impact on performance will be negligible because, by the time a TCU gets its value back from the functional unit, all other queued requests will have entered the functional unit and its queue will be empty. That means that TCUs sharing the same functional unit will get slightly out-of-synch with each other and will then not even need to pay queuing penalties for sharing the resource.

The second design choice that surprised me was the mesh-of-trees interconnection network [10] because it scales with the square of its ports, raising questions about the scalability of the design. Besides the classic modeling known as *VLSI complexity* that takes area into account (as reviewed in [10]), there are two solutions to mitigate the resource-hungry interconnect when scaling XMT to larger numbers of TCUs: one is to use a different interconnect, for example, a hybrid mesh-of-trees and butterfly interconnect [9]; the other solution is to increase the number of TCUs per cluster, thus reducing the ratio of interconnect ports to TCUs. For example, consider an XMT configuration with $N = 64$ TCUs in 8 clusters of 8 TCUs. The size of the interconnect will be proportional to the square of its ports, which is the number of clusters (i.e., $Interconnect \sim 8^2 = 64 = N$). Now consider an XMT configuration with four times more TCUs ($N = 256$) and twice as many TCUs per cluster, i.e., 16. The number of clusters and interconnect ports will be 16, and the size of the interconnect will be $Inteconnect \sim 16^2 = 256 = N$. So the design of larger XMT can be rebalanced to control the scaling of the interconnection network. Of course, this trick will only work as long as increasing the number of TCUs per cluster remains beneficial. Furthermore, note that the arbitration in mesh-of-trees

is relatively simple, resulting in area requirements that are not as excessive as the combinatorial complexity of the design would suggest [10]. More details about the interconnection network of the XMT architecture can be found in the dissertation of Aydin Balkan [11].

The final design choice was the lack of local coherent caches at the cluster or TCU level. This means that practically each memory access has to travel through the interconnection network, which has a non negligible latency. However, mechanisms such as prefetching of memory load instructions, broadcasting of read-only values, and the planned inclusion of compiler managed scratch-pads, manage to hide latency and to reduce or even eliminate memory hot-spots. In combination with the absence of cache coherence overheads, these latency reducing mechanisms allow XMT very competitive performance, while simultaneously relieving programmers from many locality headaches they would have on other architectures. In his dissertation [90], Xingzhi Wen presents a comparative study of the two alternatives: having coherent private caches or only using latency tolerating mechanisms. The results show that not having private caches on XMT is preferable in terms of performance, especially for fine-grained codes, assuming that the compiler is able to efficiently use prefetching and broadcasting to hide the latency to the shared cache. For that reason, a significant portion of George C. Caragea's dissertation [21] focused on prefetching for XMT.

2.2 Work Stealing Scheduling

Work Stealing is a distributed dynamic scheduling algorithm that has recently gained popularity for scheduling task parallel codes because of its low overhead, which supports fine-grained tasks, and because it is provably efficient in terms of time, space, and communication [16]. The central idea is that worker threads that become idle try to steal work from workers that are busy, as opposed to work sharing, where the worker that encounters additional parallelism attempts to push it on

other workers that may soon run out of work. At a high level, one of the intuitions behind work stealing is that it is beneficial to reduce the total work performed by a parallel computation at the expense of the length of the critical path. The working assumption is that the amount of parallelism is much greater than the number of workers, and, therefore, the dominant term is the total amount of work.

2.2.1 Motivation for Dynamic Scheduling

Static scheduling of parallel loops is easy: the number of tasks can be divided by the number of processors at run-time, to yield the number of tasks that each processor should execute. While this works well when the iterations of the parallel loop perform approximately the same amount of work, such as for several regular affine (dense-matrix) scientific codes, such a naive partitioning of tasks results in load imbalance and poor performance, when the iterations of the parallel loop perform unpredictable and differing amounts of work. For example, Figure 2.3 shows a parallel loop where each iteration calls a function `foo`; depending on the input, `foo` may perform vastly different amounts of computation in different tasks. Here, a dynamic scheduler is likely to achieve better load-balancing and perform better because it will allocate tasks to processors at run-time, as they become free.

```
spawn(low, high) {
    A[$] = foo($);
}
```

Figure 2.3: A Case for Dynamic Scheduling: Load Imbalance

Any dynamic scheduling method must handle both non-nested parallel loops (e.g., Figure 2.3), and nested parallel loops. Nested parallel loops arise not only in simple syntactic nesting (not shown), but in recursive parallelism as well. For example, Figure 2.4 shows the parallel code for quicksort. The quicksort routine sorts the array in the range *start* to *end*. First, the partition procedure chooses a

pivot value from the subarray $[start, end]$, places all the smaller elements than the pivot before it and all the larger elements after it, and returns the position of the pivot. Next, quicksort calls itself recursively in parallel on the two subarrays defined by the pivot. Deeply nested parallelism arises in quicksort because each recursive invocation introduces a new parallel loop level.

```
void quicksort(int A[], int start, int end) {
    int pivot = partition(A,start,end);
    spawn(0,1) {
        if($==0) quicksort(A, start, pivot);
        else     quicksort(A, pivot+1, end);
    }
}
```

Figure 2.4: Quicksort in XMTC: Example of Recursively Nested Parallelism.

Note that the most natural way to parallelize quicksort is not by using a parallel loop as shown, but using a parallel function call for one of the two recursive calls. Currently, XMTC only supports the parallel loop construct however, which can be used as shown to achieve the same result.

To support *natural programming idioms* and *ease of programming*, dynamic scheduling can be used to deliver good *performance* on any code the programmer happens to write. Indeed, the most natural and succinct way of writing quicksort is using recursively nested parallelism, which work stealing schedulers support. Many divide-and-conquer (and other) algorithms are also written most naturally with recursively nested parallelism, and should be supported. Of course, one can argue that sorting (or other divide-and-conquer algorithms) can be rewritten iteratively, or that scheduling in such codes can be handled by the programmer. Unfortunately, both options tend to greatly increase the burden on the programmer, which we are trying to alleviate.

Supporting nested parallelism (and not just outer parallelism) is essential for

performance for the following reasons. *First*, the outer parallelism – parallelism created by the original (sequential) thread – might not create enough tasks. In quicksort the outer parallelism only creates one additional task, which in most cases is not enough to feed all processors. *Second*, the outer tasks might contain vastly different amounts of computation, jeopardizing load-balance. In quicksort, depending on the pivot found by the partition procedure, the two outer tasks might sort arrays of vastly different sizes; so, if inner parallelism is serialized the potential for load-imbalance and the resulting performance degradation are lurking. Conversely, supporting nested parallelism enables the creation of more tasks by dissecting the outer tasks, and with dynamic scheduling, it leads to better load balance and, ultimately, to better performance. *Third*, a successful programming language should be *modular*, allowing the programmer to write a function once and call it from sequential or parallel contexts alike; thus, supporting nested parallelism (through function calls) is needed for modularity. These reasons make a compelling case for supporting nested-parallelism and dynamic scheduling.

2.2.2 Work Stealing Background

The idea of work stealing is at least as old as Burton et al. [18] and Halstead's [44] work on functional programming, but it started gaining popularity with Cilk [38] and is now incorporated in many commercial products [59, 76, 61, 62]. In work stealing, each worker (typically an OS thread mapped to a hardware thread or processor) that encounters parallel work starts executing some of that work and places the continuation (the remaining parallel work and the rest of the parent) on a shared work-pool. When a worker runs out of work, it searches for available work on that shared work-pool. The design of the work-pool is what makes work stealing unique: it consists of P double-ended queues, called *deques*, one for each of the P workers. They are called double-ended because data is accessed from both ends: each worker treats its own deque as a stack, accessing the deque at one end, and it

treats the other deques as queues, accessing them at the other end, when its own deque is empty.

A worker pushes parallel tasks it encounters onto its deque and pops tasks when it runs out of work, treating its own deque as a stack. When a worker runs out of work and its deque is empty, it becomes a *thief*: it picks at random another worker, the *victim*, and tries to *steal* a task from its deque. Popping the newest task from the local deque results in *depth-first execution*, and stealing the oldest task from a victim deque results in *breadth-first thefts*.

Four major benefits of work stealing are the following: *(1)* depth-first execution promotes locality by first working on one's own deque and *(2)* keeps the memory footprint under control; *(3)* breadth-first thefts tend to result in stealing larger chunks of work, thereby resulting in good load-balancing; *(4)* the deques can be implemented efficiently with low synchronization overheads. A disadvantage of work stealing is its stealing phase, when idle workers randomly probe deques for work, causing potentially unnecessary interprocessor communication.

Unlike parallel function calls, which create one new parallel task at a time, parallel loops can create multiple tasks simultaneously, allowing crucial optimizations: all the tasks originating from a parallel loop invocation can be packed into a single *task descriptor* (TD) by specifying their range, and they can be distributed to workers as needed by splitting the TD. Thereafter, the scheduler can decide at run-time when to stop splitting TDs, to avoid unnecessary and expensive deque operations, which typically require expensive memory-fences. The compiler can also estimate the cost of very short iterations and decide to combine them into fewer, longer iterations, to amortize scheduling costs. Using parallel function calls hides the logically simultaneous creation of parallelism from the compiler and the runtime system, thus disabling the above optimizations and leaving the programmer responsible for keeping parallelism somewhat more coarse-grained in order to reduce the scheduling overheads.

TDs, also known as work descriptors, are used to describe ranges of tasks coming from parallel loops, reducers, or other multi-argument operations. The specific structure of TDs is implementation specific, but one possible implementation is the following: the ID of the first task and the number of tasks (or the ID of the last task) can be used to represent the range; a single pointer to the code to be executed is necessary since all tasks execute the same code using the task ID (iteration ID) as a parameter; a pointer to the stack frame of the parent task is also needed to allow access to its variables and keeping track of the number of pending child tasks. Optionally, TDs may contain additional fields such as a *grain-size*, the number of *chunks* into which to split the TD, or a cost estimate of the tasks it contains.

Another reason parallel loops deserve direct support is that recreating them with parallel function calls is inconvenient and inefficient. To do so, the programmer must either write a sequential loop with a parallel function call in its body, or implement the creation of parallelism recursively, using a divide-and-conquer approach. The first approach leads to serialized creation or parallelism and a memory footprint that is linear in the number of tasks. Moreover, the performance will be very poor for fine-grained tasks because the overheads of creating and scheduling them will outweigh the benefit of parallelism. The second approach leads to parallel (fast) creation of parallelism in a binary tree, but it is tedious and potentially error-prone for the programmer. A work stealer for parallel loops should automate the second solution (divide-and-conquer) and not hide from the compiler the simultaneous creation of parallel tasks. The same argument applies to supporting other multi-argument parallel constructs, such as sum-like reducers and scan operations.

When it comes to scheduling TDs, there are several different ways to treat them. The two main categories include recursively splitting the TD range and iteratively breaking off constant-sized chunks. We cover those alternatives in the next subsections.

2.2.2.1 Serializing Work Stealing (SWS): Work-First

The *work-first* approach, which we call *serializing work stealing* (SWS), keeps the first *grain* iterations of a TD and pushes the rest onto the local deque. The drawback of this approach is that a TD created by a parallel loop is never split, so accesses to it by workers contending for work will be serialized.

We illustrate with a simple example how serializing work stealing works with TDs, and discuss its shortcomings next. Assume worker A encounters a parallel loop with 16 iterations; A will create a TD with iterations 2 through 16, place it on its deque, and start executing the first iteration. For simplicity, assume those iterations do not create nested parallelism. In the mean-time, worker B steals the TD from A's deque, takes iteration 2, places the remaining TD (iterations 3-16) on its deque, and starts executing iteration 2. A eventually finishes executing iteration 1, looks for work on its deque, which it finds empty, so it tries to steal work from B; it is successful, takes iteration 3, places the remaining TD on its deque, and starts executing iteration 3.

This example illustrates four shortcomings:

1. If two or more workers end up executing tasks from a TD, they will keep stealing the TD from each-other, effectively serializing accesses to it.

2. On modern multicores, thefts are expensive because they induce coherence traffic by modifying remote deques, which presumably reside in the private cache of the victim worker.

3. Unless a grain-size is provided, each time a worker needs more work, it removes a single iteration from a TD; this means that TDs (and thus deques) will be accessed as many times as the tasks they have, which, for fine-grain iterations, introduces significant overheads.

4. Because of the implicit barrier at the end of parallel loops, tasks need to syn-

chronize upon termination, usually by atomically decreasing a variable representing the number of pending tasks. Unless a grain-size is provided, iterations are executed one-at-a-time, and synchronization will also happen individually for each task, possibly inducing significant overheads.

2.2.2.2 Help-First Work Stealing

The *help-first* work stealing approach treats a parallel loop as a sequential loop whose iterations each spawn one parallel task. This approach also serializes the creation of the tasks, but it creates a TD per task, allowing parallel access to them, unlike the work-first approach described earlier. However, by creating a TD per iteration, or per k iterations when a grain-size of k is provided, help-first work stealing ends up having a potentially unbounded memory footprint relative to the sequential footprint for the same code.

Guo et al. [43] have implemented a scheduler that adaptively switches between the help-first and work-first work stealing to get the benefits of help-first task creation, while keeping the memory footprint bounded. However, the depth (critical path) of scheduling a parallel loop of N tasks is linear in N for both help-first and work-first approaches. This depth is added to the critical path of the application and, in some cases can overwhelm it. The eager binary splitting approaches below reduce this depth from linear to logarithmic. Furthermore, the work by Hendler et al. [47] seems to suggest that recursive splitting benefits performance by spreading tasks around in larger chunks, rather than stealing a single task at a time.

2.2.2.3 Eager Binary Splitting (SP & AP)

Intel's Threading Building Blocks[1], Cilk++ [62] and CilkPlus implement a *Eager Binary-Splitting* (EBS) work stealing schedulers: upon creating, stealing, or popping a TD, a worker splits it into two TDs of approximately equal numbers of iterations and pushes one on its deque; then, it continues splitting the remaining TD

until some threshold. EBS is *eager* because splitting proceeds regardless of run-time conditions such as load.

An important performance consideration for eager binary splitting is when to stop splitting. While splitting TDs to create enough parallelism and to load-balance is crucial, excessive splitting induces unnecessary overheads, which can hurt performance. It can be preferable to coarsen parallelism by stopping the splitting of TDs before they are reduced to a single iteration and execute all the iterations in the coarser TDs sequentially. Finding this *stop-splitting threshold (sst)* is hard because it depends on several factors, such as the number of available hardware threads (processors), the number of tasks of each parallel loop (which can be a function of the size of the input), and the calling context.

TBB offers two options for controlling that threshold: simple-partitioner (SP) and auto-partitioner (AP). Cilk++ and CilkPlus only implement simple-partitioner. Cilk++ has a mechanism inherited from Cilk[38] for reducing parallelism overheads by creating two versions of functions and choosing at run-time which one to execute: the one for fast local and serialized execution with simplified synchronizations, or the one for true parallel execution. This mechanism is orthogonal to our proposed lazy-scheduling and combining the two approaches would be beneficial.

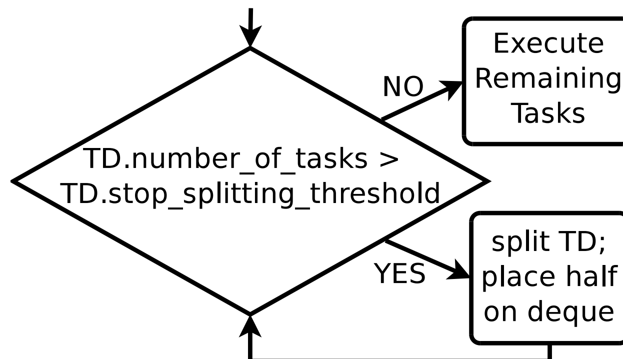


Figure 2.5: Processing a Task Descriptor with Simple-Partitioner.

Simple-Partitioner. Figure 2.5 shows how simple-partitioner splits a TD while the number of iterations in its sub-range is above a stop-splitting-threshold, referred to as *grain-size* in TBB's manual [1]. This eagerness to split may result in an excessive number of TDs being created, which is why the programmer is expected to define an appropriate *sst* to stop the splitting earlier. The TBB manual [1] suggests the following approach to determine the appropriate *sst*:

1. Set the stop-splitting-threshold parameter of the parallel loop to 10,000. This value is high enough to amortize scheduler overhead sufficiently for practically all loop bodies, but may unnecessarily limit parallelism.

2. Run your algorithm on one processor.

3. Start halving the threshold parameter and see how much the algorithm slows down as the value decreases.

⇒ A slowdown of about 5-10% is a good setting for most purposes.

There are two problems with this approach. *First*, it is extremely tedious. Not only does the programmer have to provide a threshold, they have to run their program several times to find the appropriate threshold. Moreover, if the code has multiple parallel loops, a different threshold has to be determined for each loop, which means more runs. Ideally we would want the 5 to 10% slowdown to be only compared to the code of the parallel loop, not of the whole application, so the programmer will have to isolate the parallel loops during this tuning process and time them separately. Finally, because the code will run on a single processor, this tuning process will also be very slow. *Second*, another equally serious problem is that the resulting fixed threshold limits the performance portability of the code to different platforms, inputs and contexts, as mentioned in the introduction. We provide more evidence that manual coarsening is tedious and harms performance portability in Chapter 3.

In conclusion, eager binary splitting with simple-partitioner is an improvement over work-first serializing work stealing because splitting TDs solves the problem of serialized access to the tasks of a parallel loop, but determining the grain-size (sst) manually is very tedious, and, if it is a fixed constant as suggested by TBB's tuning procedure, it harms performance portability. Work-first and help-first work stealing also have the same issue with the grain-size.

Auto-Partitioner. TBB's other option for controlling splitting, auto-partitioner (AP), splits the tasks of a parallel loop into $K \cdot P$ TDs, assuming the number of iterations in the original parallel loop is at least $K \cdot P$, where P is the number of workers and K is a small implementation-specific constant. Auto-partitioner was recently chosen as TBB's default scheduler because it relieves the programmer from manually picking the *sst* and delivers good performance. Auto-partitioner has two fixed parameters, K and V, as well as an additional *chunks* field per TD (called n in [76]). When executing a parallel loop and creating its TD, *chunks* is initialized to $K \cdot P$. Every time the TD is split, *chunks* is also halved, and whenever a TD is stolen, *chunks* is set to be at least V, which gives auto-partitioner some limited run-time granularity adaptivity. A TD is not split further if $chunks \leq 1$ or if it is not divisible (e.g., contains a single task). K and V are set to four in [76].

Instead of coarsening parallelism by combining iterations using the *sst*, auto-partitioner uses *chunks* to determine into how many pieces to split a TD. This is preferable because it does not require programmer tuning, allows for some platform and dataset portability (but still not context portability), and performs better than simple-partitioner in most cases. The *sst* is still available to the programmer, in case more aggressive coarsening is required. For example, if the iterations of a parallel loop are few and fine-grained, auto-partitioner might still perform excessive splitting.

The lack of context portability in auto-partitioner is a serious problem. While

splitting iterations into $K \cdot P$ TDs for a parallel loop executed from the original sequential thread is usually a good heuristic, if that same loop is executed in a nested context, the outer parallelism will likely suffice and fewer chunks would be preferable. For example, for d levels of nested parallel loops of N iterations each, auto-partitioner will create over the course of the execution $N^{d-1} \cdot (K \cdot P)$ TDs for the most deeply nested loop, which may be excessive. These TDs will not be simultaneously available in memory, but the overheads of creating them over the course of the execution is still substantial. The maximum number of TDs concurrently present in the system will be $O\big(P \cdot \log(K \cdot P) \cdot (d-1)\big)$. Reducing K to reduce the number of chunks may result in insufficient splitting (and parallelism) for non-nested loops, so it is not a viable solution, and the lack of context portability seems to be inherent to auto-partitioner. Our Lazy Scheduling approach overcomes the portability pitfalls of SP and AP and the serialization of parallelism creation of work-first and help-first work stealing, without requiring programmer tuning.

Another potential danger with auto-partitioner, even without nested parallelism, is that once it starts executing one of the original $K \cdot P$ "fat" chunks, it will execute it to completion, without the possibility of revisiting the coarsening decision of not further splitting the TD. If the tasks of a loop are severely imbalanced, one of the "fat" chunks may contain most of the work and the performance will suffer of poor load balancing. For the same reason, the time bound (presented below) for a work stealing schedule does not apply to auto-partitioner.

2.2.3 Theoretical Bounds

Blumofe et. al [16] helped the adoption of work stealing by proving the good performance of randomized work stealing for fully-strict computations using only parallel function calls. The expected time to execute a fully strict computation on P workers using randomized work stealing is $T_1/P + O\big(T_\infty\big)$ where T_1 is the minimum sequential execution time, i.e., the total work, and T_∞ the minimum execution time

with an infinite number of workers, i.e., the depth of the parallel computation (the length of the critical path). The space required is at most PS_1, where S_1 is the minimum space requirement for the sequential execution.

More recent results relax the restriction of fully-strict computations but, to the best of our knowledge, still omit to include language constructs that introduce multiple tasks simultaneously, such as parallel loops. An exception is [29] (Chapter 27), which talks about the added $\log N$ term on the critical path for simple-partitioner. In the presence of loops, the above bounds need to be amended as we will discuss in Section 4.10.

Chapter 3

Coarsening

In this chapter, we tackle the problem of coarsening the abundant parallelism coming from declarative code. We argue that manual coarsening either leads to performance portability issues or requires substantial expertise from a programmer, and we show the sensitivity of performance to coarsening. We identify two goals of coarsening and propose achieving each of the two goals using separate techniques in order to preserve performance portability and to maximize performance, whilst relieving the programmer from manual coarsening, at least partially.

Current implementations of languages that support nested task-parallelism (OpenMP, Cilk, TBB, TPL, ...) warn programmers not to overexpose parallelism to avoid excessive scheduling overheads (e.g., [60]). An earlier version of the TBB Manual [1] suggested that, for each parallel loop of the program, the programmer should pick an appropriate *grain-size* describing when a range of tasks should no longer be split for parallel execution. This grain-size parameter coarsens parallelism by effectively combining *grain* tasks of a parallel construct into a single task. As elaborated in Section 2.2.2.3, the suggested procedure for determining the grain-size was to repeatedly run the parallel construct on a single thread using decreasing values for the grain-size, until the overhead reached about 10% of the sequential execution of that construct. Such a procedure is not only tedious, but leads to code that is not performance portable as we will show in Section 3.3, later in this chapter.

The current version of the TBB Manual omits this section on picking a grain-size because TBB has since switched to *auto-partitioner* as its default scheduler, which performs adequate coarsening for most codes that do not have nested parallelism. However, as we argued in Section 2.2.2.3, auto-partitioner is not adequate for declarative codes with deep nesting, such as recursively nested codes used in

graph traversals and state-space exploration, because it fails to perform sufficient coarsening, due to its inability to adapt to different calling contexts.

For someone working in high performance computing it may be counter-intuitive why coarsening should be a serious problem for general purpose code since, in that domain, it is not. The first reason why coarsening is not a serious problem in high performance computing is that it has focused on codes with flat or relatively flat parallelism, as exemplified by the codes in the NAS benchmark suite [8]. For such codes, coarsening is indeed not very challenging. Another scenario where coarsening is not a serious problem is when the programmer has good command of the code of the entire application and knows the characteristics of its inputs and of the target platform. In that case, the programmer generally knows when enough parallelism has been exposed for the target platform and can coarsen the rest.

Algorithm 3.1 Fully Parallel Matrix Multiplication

```
1: INPUT A_{N×K}, B_{K×M}
2: OUTPUT R_{N×M}
3: for all i ∈ {1, ..., N} do
4:     for all j ∈ {1, ..., M} do
5:         R[i][j] ← 0
6:         for all k ∈ {1, ..., K} do reduction (+ : R[i][j])
7:             R[i][j] += A[i][k] · B[k][j]
8:         end for
9:     end for
10: end for
```

On the contrary, for general purpose parallel programming we treat the input and the number of processors as unknowns. Now, it is no longer clear how much parallelism to expose in a code as simple as matrix multiplication. Algorithm 3.1 shows that each multiplication can be done in parallel by using a parallel reduction for the innermost (sequential) loop (Line 6). In this case, the reduction operation[1] is a parallel summation for each $R[i][j] = \sum_{k=1}^{K} A[i][k] \cdot B[k][j]$. If N and M are

[1]The notation **reduction** $(+ : var)$ is taken from OpenMP, and it informs the compiler that the aggregation $(+)$ of values into the memory location var must happen safely (e.g., atomically) in parallel.

too small to provide enough parallelism, as in the extreme example of multiplying two vectors ($N = M = 1$), then parallelizing the innermost loop may be profitable. On the other hand, the overheads of parallelizing it will penalize the performance of most invocations where N and M create enough parallelism. But, to make things harder, we also treat the number of processors as an unknown, which means we cannot really answer whether enough parallelism has been created. Even if we restrict ourselves to the range of 4 to 1000 threads (Intel's i3, to nVidia's Fermi GPU), and expose enough parallelism for 1000 threads, chances are we will pay a noticeable performance penalty on the smaller machines.

Algorithm 3.2 Matrix Multiplication: Parametrically Coarsened

```
1:  grain_i ← max(1, C/(K·M), N/(4·P))
2:  for all i ∈ {1, ..., N} with grain = grain_i do
3:      grain_j ← max(1, C/K, (M·N)/(4·P))
4:      for all j ∈ {1, ..., M} with grain = grain_j do
5:          R[i][j] ← 0
6:          grain_k ← max(C, (K·M·N)/(4·P))
7:          for all k ∈ {1, ..., K} with grain = grain_k do reduction (+ : R[i][j])
8:              R[i][j]+ = A[i][k] · B[k][j]
9:          end for
10:     end for
11: end for
```

An other option would be for the programmer to write code that dynamically makes coarsening decisions at runtime, based on the input and the number of workers of the target platform. An example is shown in Algorithm 3.2 where we want to create four times as many tasks as the number of workers ($4 \cdot P$) to get good load balancing, and each task should perform at least C multiplications to amortize scheduling overheads. Despite the simplicity of these two requirements (maximum number of tasks and minimum number of multiplications per task) and despite the code being regular, which greatly simplifies enforcing them, the parametrically coarsened code is not as simple anymore.

While such parametric coarsening can improve performance, it can still create

an excessive number of tasks and severely hurt performance in two scenarios. The most common is when code is called from a parallel context. If, for example, each worker calls Algorithm 3.2 in parallel, the total number of tasks created will be $4 \cdot P^2$. In general, the number of tasks can grow exponentially with the recursive depth. An other scenario is when the operating system reclaims some of the processors that it had previously granted a parallel application, in which case the application should react by creating fewer tasks. For example, Pan et. al [74] present a framework for the operating system to increase or decrease the number of worker threads of a task parallel application. They also address the problem of composing code from different parallel libraries that assume they can each create P worker threads (such as the TBB library) and that would oversubscribe the machine resources by initiating too many workers. This is an important part of supporting general-purpose parallelism, but does not address the need for coarsening excessive parallelism on the application side.

The rest of this chapter is structured as follows: we refine what types of coarsening we will consider automating and why; for example automatically changing the underlying algorithm is beyond the scope of this work. Within these constraints, we present the two goals that coarsening has to achieve: creating enough tasks for good load balancing while keeping tasks coarse enough to avoid excessive scheduling overheads. This concept was also present in the parametric coarsening example of Algorithm 3.2. Since we propose that the programmer should not coarsen parallelism, we proceed to discuss the different *stages* through which available task parallelism goes, from code to execution and present some guidelines as to where each goal of coarsening can be achieved as well as a high level view of the coarsening performed by the various schedulers we compare in this dissertation. Then, we present evidence that performance is sensitive to coarsening, when it is not done in a parametric way. As far as parametric coarsening goes, we believe that it is something general-purpose programmers should not have to do, especially since it is not

context sensitive and it is challenging for irregular applications (e.g., [49]). We then define two metrics for evaluating the sensitivity of a code to varying environments (inputs, platforms, and contexts), and we propose to use these metrics to quantify performance portability. We show that static coarsening fails to be performance portable, and even parametric coarsening is not as efficient as one might expect. Finally, we present two compiler optimizations implemented in XMTC that perform static and dynamic coarsening for extremely fine-grained tasks.

3.1 Characterizing Coarsening

We focus on three types of coarsening within the realm of task-parallel programming languages: *(1)* picking a grain-size G for a parallel construct (e.g., loop or reducer) indicating that parallel tasks should contain at least G iterations during execution; *(2)* not parallelizing a computation, for example coding a parallel loop as a sequential loop; *(3)* explicitly serializing part of the computation (e.g., parallelism cut-off discussed below in Algorithm 3.3). Coarsening that involves changing the algorithm is beyond our scope. Within this narrower definition for coarsening task-parallelism we identify two goals that coarsening has to achieve.

The two goals of manual coarsening of task-parallel code are (1) to ***amortize*** scheduling overheads by increasing the granularity of extremely fine-grained tasks, and (2) to ***prune*** the exposed parallelism to minimize the wasted overheads of deploying too much parallelism, without underexposing parallelism which could hurt performance. In this section, we argue that static coarsening (manual or automatic) is adequate for amortizing overheads, but inadequate for pruning parallelism.

Understandably, the two goals of coarsening are not decoupled. Amortizing scheduling overheads prunes parallelism, but may still leave an excessive amount of parallelism exposed. Symmetrically, pruning parallelism also increases granularity, but, if there is not much parallelism, it may fail to amortize scheduling overheads. To make things harder, sometimes the same technique can be used to achieve either

Algorithm 3.3 Queens pseudocode. $depth \in [1, N]$

```
 1: procedure QUEENS(N, partial_sol, depth)
 2:     for all i ∈ [1, N] do
 3:         if OK_TO_ADD(i, partial_sol, depth) then
 4:             append i to partial_sol
 5:             if depth < N then                                  ▷ Recursion
 6:                 if depth > CUTOFF-DEPTH then
 7:                     QUEENS-sequential(N, partial_sol, depth + 1)
 8:                 else
 9:                     QUEENS(N, partial_sol, depth + 1)
10:                 end if
11:             else                                                ▷ Found a Solution
12:                 atomic(global_sol_count+ = 1)
13:             end if
14:         end if
15:     end for
16: end procedure
```

of the goals. For example, Algorithm 3.3 shows the pseudocode for counting the ways to place N queens on an N by N chessboard, without them attacking each other. The function has recursively nested parallelism, as it has a parallel for-loop and it calls itself recursively from within that loop. The first argument is the size of the board N, the second is the partial solution computed so far (originally the empty set), and the third argument is the depth of the recursion, starting at 1. Each recursive invocation of the procedure tries to place a queen on each column of the $depth^{th}$ row in parallel. If doing so does not conflict with the partial solution so far (Line 3), the queen is added to it (Line 4), and the procedure is called recursively for the next row ($depth + 1$) (Line 7 or 9), unless all rows have queens, in which case a solution was found and the global solution counter is atomically incremented (Line 12).

We can limit the parallelism of the *QUEENS* computation by choosing between a parallel invocation of $QUEENS$ and a sequential one (lines 6-10), based on the recursive depth. This technique is known as *parallelism cut-off*. If our intention is to amortize fine-grained tasks, we should cut-off the last α recursive levels (for some value of α), when $depth > N - \alpha$, whereas if our intention is to prune the amount

of exposed parallelism, we should cut-off after the first R recursive levels (for some value of R), when $depth > R$.

Figure 3.1 distinguishes visually the two goals of coarsening. If you visualize the computation as an N-ary tree where the distance of a node from the root is its recursive depth, to amortize the overhead per task, you group nodes starting from the leaves, but to to prune parallelism, you only allow the first few levels of the tree to unfold.

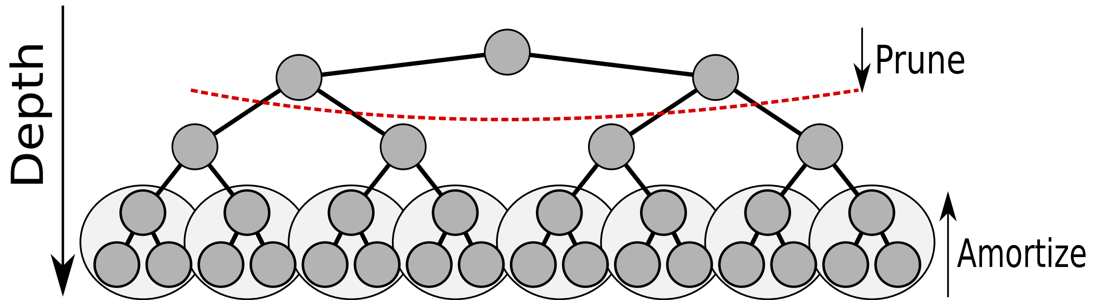


Figure 3.1: Amortizing Overheads vs. Pruning Parallelism

Before delving into the specifics of which techniques to use for achieving each of the two goals, one fair question is why should pruning parallelism be necessary if we properly amortize overheads by coarsening very fine-grained tasks. In QUEENS, grouping subtrees near the leaves amortizes the overheads per task, but to reach those coarsened tasks, the computation has to traverse the whole tree, paying the scheduling overhead multiple times, once for each edge traversal. In effect we have not amortized the scheduling costs in the middle of the tree, which creates the need for pruning parallelism.

The reason for not coarsening in the middle of the tree is that it changes its shape and therefore the algorithm. The challenge is to make all of the outgoing edges (i.e., subtasks) of a coarsened subtree available to the scheduler before proceeding with the execution of one of them. Failing to achieve that may result in significant parts of the tree being unavailable for execution causing scaling issues similar to the ones we will present in Section 4.5. Effectively, while the execution order of

the overall tree is depth-first, internally the execution of the coarsened mid-tree nodes should happen in breadth-first order. Moreover, the aggregate scheduling overheads of all the outgoing edges should be reduced, because just saving the overheads of the internal edges of a coarsened sub-tree will only lead to small savings. Besides the fact that changing the traversal order to breadth-first mid-tree requires changing the algorithm, which is not trivial, and that we do not currently have a good way to aggregate task creation and synchronization coming from multiple call-sites, breadth-first execution also substantially increases the memory requirements of applications. For these reasons, we will not consider mid-tree coarsening as a general coarsening technique, although it may be useful in certain circumstances.

Deciding how much to coarsen extremely fine-grained tasks to *amortize* scheduling costs is relatively easy, because the overhead per-task (OPT) of modern schedulers is very low, which makes coarsening necessary only for extremely fine-grained tasks, and because such tasks are generally short and simple. Moreover, the amount of coarsening only depends on the overhead per task, which can be approximated by a constant upper-bound (e.g. 100 cycles), and on the work per fine-grained task, which can be typically estimated at compile-time.

Amortizing the overhead per task must happen before the runtime tries to schedule the tasks because its goal is to reduce the frequency of calls to the scheduler. Eventually, a mature compiler may be able to completely relieve the programmer from this coarsening task. This is desirable because the overhead per task can depend on the platform, affecting the amount of coarsening needed. However, we expect the cross-platform variation of the OPT to be small enough to allow manual coarsening without harming performance-portability. Thus, it is reasonable for programmers to perform this simpler type of coarsening, while compilers make progress towards automating it. Section 3.5 presents the static coarsening passes available in the XMTC compiler.

Using static coarsening to decide how much to *prune* the exposed parallelism is

unnatural, however, because the amount of parallelism is often unknown at compile-time (or at programming-time). The amount of parallelism exposed by a module (e.g., a function) may depend on the size or shape of the input D. The module may be called from an execution point where a lot of parallelism can be exploited, such as a sequential section, or from a point where the system is saturated, so the amount of parallelism needed depends on the context C. Moreover the number of cores available for parallel execution is, at best, a parameter for general purpose code aiming to target multiple parallel platforms, and, more likely, a dynamically changing value on modern multiprogrammed systems, where the operating system may shift computing resources from one process to another. Static coarsening cannot adapt to the dynamic nature of these parameters (D, C, and M) and typically results in overfitting to a small range of these parameters. Outside that range, the performance of the code can be far from optimal and, hence, not performance portable. For some domains, such as HPC, where the goal is to get the best possible performance on a specific machine with inputs of a given size, overfitting is not a drawback and getting the best possible performance is preferred, but for general-purpose programming, this is undesirable.

Instead of the inadequate static coarsening currently used, *we propose that the run-time system, which has access to some dynamic load information during execution, should assume the responsibility of adaptively deploying the exposed parallelism as needed*. Our proposed lazy scheduling approach significantly reduces the wasted overheads of deploying declarative parallelism compared to existing state of the art work-stealing schedulers, and it offers a scalable alternative to manual coarsening, while preserving performance-portability.

3.2 Stages of Parallelism

In this section, we define four *stages* of parallelism to help structure the understanding of the compilation and execution of task-parallel code. We call the first

stage *Code Parallelism (CP)*. It is the parallelism exposed by the programmer in their parallel code. The code parallelism is what the compiler takes as its input. Through static analysis and transformations, the compiler can coarsen the code parallelism, for example by combining iterations of a parallel loop, and it may also parallelize code that was originally sequential in the program. Generally speaking, the executable produced by the compiler exposes a different collection of parallel tasks to the run-time than the code parallelism. We call this second stage *Executable Parallelism (EP)*. During execution, the run-time (e.g., the scheduler or just-in-time compiler) has access to all of the Executable Parallelism, but it may choose to coarsen some of it or to only deploy part of it on its shared work-pool to reduce overheads. We call the tasks that are placed in the work-pool *Deployed Parallelism (DP)*, even if they eventually execute locally, and the tasks that bypass the work-pool and are executed locally instead *Pruned Parallelism (PP)*. By definition, the following relations always hold, where $\setminus$ is the standard notation for set subtraction:

$$DP \cup PP = EP, \quad DP \cap PP = \emptyset, \quad PP = EP \setminus DP$$

Traditional work stealing (e.g., Cilk[38], simple-partitioner in TBB, and TPL) deploys all of the executable parallelism for parallel execution by placing it onto the deques. Furthermore, the compiler does not transform parallelism, and therefore, the code parallelism coincides with the executable parallelism ($CP = EP = DP$, $PP = \emptyset$). Note that for a parallel loop of N iterations, traditional work-stealing will only expose $N-1$ of them on the deque, but we still consider PP to be empty, and for simplicity, $CP = EP = DP = N$ (instead of $N-1$). Also, if the programmer-defined grain-size of that loop is g, then $CP = EP = DP = N/g$.

TBB's auto-partitioner initially coarsens the tasks of a parallel construct (e.g., loop, reducer) into $K \cdot P$ *chunks*, where K is a small constant ($K = 4$ in TBB's implementation) and P is the number of worker threads, but it may later undo

some of the coarsening to improve load balancing. Therefore, $DP = EP = CP$, and $PP = \emptyset$ since all the work is exposed on the work-pool, modulo the chunking which acts almost as a grain-size.

Conversely, our lazy scheduling [83] follows our proposed separation of coarsening duties: the compiler and the programmer are responsible for increasing task granularity to amortize overheads, and the run-time is entirely responsible for pruning the deployed parallelism by only exposing a limited amount of parallelism on the work-pool based on load conditions.

$$DP \subseteq EP \subseteq CP, \text{ and } PP = EP \setminus DP$$

The goal of lazy scheduling is to minimize DP (maximize PP) without adversely affecting performance. In other words, the goal is not to deploy too much or too little parallelism on the work-pool, but just the right amount (also known as *the Goldilocks problem* [13]).

The separation of coarsening responsibilities gives lazy scheduling a *productivity* advantage over traditional work-stealing and auto-partitioner, by lifting from the programmer the burden of coarsening to *prune* parallelism and part of the burden of *amortizing* scheduling overheads, while at the same time preserving performance-portability. In the next section, we will show the sensitivity of performance to manual coarsening, which testifies to our claim that manual coarsening can hurt performance portability.

3.3 Sensitivity of Performance to Coarsening

In this section, we present two experiments that demonstrate how manual coarsening can lead to sub-optimal code that is not performance portable. In the first experiment, we follow the manual tuning procedure proposed in TBB's manual [1] and show that, when we run the manually coarsened code on a different input,

the performance is sub-optimal. We repeat this experiment over our benchmark suite and run the codes on XMT which gives the codes the benefit of its hardware scheduler for outer parallelism. Even with the added hardware, the performance takes a noticeable hit from the inadequate manual coarsening. In the second experiment, we take the familiar computation of QUEENS (Algorithm 3.3), and we show that the optimal coarsening depth significantly varies with the size of the input (N) and the number of available workers on the target platform. We picked QUEENS because it appears to be trivial to coarsen at first, but it actually proves to be somewhat tricky.

3.3.1 Sensitivity of TBB's proposed manual coarsening

Below we list once again the procedure proposed by TBB [1] to select the a grain-sizes when using the simple-partitioner (i.e., recursively splitting work-stealing):

1. Set the grain-size parameter of the parallel loop to 10,000. This value is high enough to amortize scheduler overhead sufficiently for practically all loop bodies, but may unnecessarily limit parallelism.

2. Run your algorithm on one processor.

3. Start halving the threshold parameter and see how much the algorithm slows down as the value decreases.

⇒ A slowdown of about 5-10% is a good setting for most purposes.

There are several problems with this approach, which is probably why it has been removed from the latest TBB manual. However, coarsening remains an issue the programmer needs to worry about even with TBB's new default scheduler, auto-partitioner. Below we repeat the issues with TBB's coarsening approach.

The first and most obvious issue with the above procedure for finding grain-sizes is that it requires multiple executions to converge to a good value for the grain-size. This is obviously tedious, but also needs to be performed separately for each parallel construct that creates multiple tasks simultaneously (e.g., loops, reducers, scans). Bergstrom et al. [13] show that the parallel efficiency, defined as the speedup over the number of workers, is sensitive to the choice of grain-size and that, not surprisingly, picking any single grain-size for all parallel constructs in their benchmark suite did not yield good overall efficiency (Figure 3 in [13]). Therefore, the programmer should pick a grain-size for each parallel construct independently by repeating the above iterative procedure.

Secondly, if the programmer is trying to solve a large problem faster by parallelizing it, it is unreasonable for them to run it multiple times on a single processor. What they may do is to use a much smaller instance of the problem to tune the code and pick the grain-sizes, and then run the code on the large input with the grain-sizes they selected for the small input. This is the premise of our experiment.

We will use a *training-dataset* and the procedure suggested by TBB to compute the grain-size for each parallel loop in our benchmarks. Then we will use those grain-sizes to execute an *execution-dataset* using TBB's simple-partitioner. Those results represent the performance achieved by the typical tuning procedure. For comparison, we follow TBB's suggested procedure to compute a different set of grain-sizes, this time using the execution-dataset. We also execute our benchmarks using this second set of grain-sizes and the difference in performance between the two represents a loss of performance-portability due to overfitting to the training dataset.

3.3.1.1 Benchmarks

Our benchmarks are summarized in Table 3.1. We ran our comparisons on a set of 8 benchmarks chosen to have various computation and communication patterns

as recommended by Asanovic et al. [7]. All benchmarks are coded in the most natural way, which is in line with our goal to provide good performance for natural programming idioms.

MM is a straight-forward dense matrix by matrix multiplication with N^3 work, N^2 parallelism (each element of the resulting array is computed in parallel), and $O(N)$ work per task. **CONV** is an $N \times N$ image by $M \times M$ filter convolution with N^2 parallelism and M^2 work per task. **FW** is the Floyd-Warshall all-pairs shortest path algorithm; the graph is represented by weighted $N \times N$ adjacency matrix. There is N^2 parallelism and constant work per task. **QSort** is quick-sort. Sub-arrays of size 100 or less are sorted using sequential quicksort. **BFS** is a breadth first traversal of a graph $G(V, E)$ given in incidence lists and with the degree of each vertex being given; given a start vertex, a level is assigned to all vertices. A pseudocode for **BFS** is given much later in Figure 6.1. Each task contains constant work. **SpMV** is a sparse matrix by dense vector multiplication. There is as much parallelism as the number of non-zero elements of the sparse array, and each task performs constant work **QUEENS** finds *all* possible solutions to placing N queens on an $N \times N$ chess-board so that no two queens can attack each-other. Algorithm 3.3 presented earlier shows the pseudocode. **TSP**, the Traveling Salesperson Problem, is the well known NP-Complete problem of finding the shortest cyclic path that visits each vertex exactly once. Just like **QUEENS**, **TSP** is also recursively nested. To perform more meaningful comparisons, we implemented TSP using exhaustive search rather than taking branch-and-bound shortcuts which can benefit unpredictably from an unrelated task scheduler decision.

For this experiment, we did not limit ourselves to declarative codes because we wanted to show the performance sensitivity to coarsening with codes typically used with TBB and other existing work-stealing schedulers. For example, *TSP*, *QUEENS*, and *QSort* have recursively nested parallelism, which is amenable to parallelism cut-off [34] (i.e., deciding to call a serial version of the recursive function

Name	Description	DOP	Work/Task
MM	Dense Matrix Multiplication	N^2	N
CONV	N^2 image by M^2 filter convolution	N^2	M^2
FW	Floyd-Warshall all-pairs shortest path	N^2	1
QSort	Quicksort	$N/100$	QSort(100)
BFS	Breadth First Search	$O(\frac{\lvert E\rvert}{Diameter})$	O(1)
SpMV	Sparse Matrix by Vector Multiplication	$N = \#non\text{-}zero$	1
TSP	Travelling Salesperson Problem	$O(\frac{N!}{(N/2)!})$	$O(\frac{N}{2}!)$
QUEENS	placing N queens on an NxN board	$O(\frac{N!}{(N/2)!})$	$O(\frac{N}{2}!)$

Table 3.1: Summary of XMTC Benchmarks

from recursive depths greater than a threshold T). For *TSP* and *QUEENS* we set the cut-off threshold to $T = N/2$. For *QSort*, instead of using the depth of the recursion, we use the size of the sub-array to be sorted or partitioned to determine the cut-off threshold: we call a serial quicksort when the subarray has less than 100 elements. We always perform the `partition` sub-routine of QSort sequentially, because we did not get a performance advantage from parallelizing it when using simple-partitioner. With Lazy Scheduling (Chapter 4), we used a parallel implementation of `partition` and call the sequential partition when the subarray has less than $T = 3\frac{N}{\#Procs}$ elements. This threshold is more complicated because the parallel partition code performs almost three times more work than the serial version; so we want to call the parallel version only when several processors are likely to be idle, such as at the onset of execution.

The only benchmarks that are declarative are **FW**, **BFS**, and **SpMV**. For **FW**, the parallelism is regular: each of the N^2 elements of the adjacency matrix can be updated concurrently; thus it made sense to expose all parallelism. With **BFS**, the outer parallelism may not be sufficient to provide enough parallelism or good load balance, so the inner parallelism is also exposed (see Figure 6.1 much later). Similarly, with **SpMV** the outer parallelism may be imbalanced or insufficient,

which justifies exposing the inner parallelism as well.

	Training Set		**Execution Set**	
Name	**Size**	**grain**	**Size**	**grain**
MM	64x64	4	512x512	1
CONV	64^2 image, 16^2 filter	1	$1K^2$ image, 16^2 filter	1
FW	64 nodes	32	512 nodes	64
QSort	10K	16	1M	256
BFS	G(10K,200K)	16	G(10K,8M)	64
SpMV	30Kx100, 60K non-zero	4	80Kx5K, 40M non-zero	64
TSP	9 nodes	1	11 nodes	1
QUEENS	N=9	4	N=11	1

Table 3.2: Benchmarks, Datasets, and Grain-Sizes.

Table 3.2 describes the training and execution datasets used for our experiment as well as the grain-sizes computed for each dataset. The smaller dataset is chosen as the training set since typically programmers will use a smaller dataset for training, given the time consuming and tedious nature of this training.

Our execution platform is the 64-TCU XMT FPGA prototype, which mean that the outer parallelism of the benchmarks benefited from XMT's hardware scheduling. Since TBB's simple-partitioner does not adapt to context, the hardware scheduler helps it by offering efficient fine-grained scheduling of outer parallelism. Simple-partitioner's excessive splitting overheads are only payed for nested parallelism. Nevertheless, even with the hardware assistance, TBB's procedure for picking the grain-size (*sst*) results in overfitting to the training input.

3.3.1.2 Results

First, we use TBB's procedure for determining the grain-size using the training set for each benchmark, then repeat the procedure using the execution set. Note

that the grain-sizes we get for the two sets (see Table 3.2) are different. More interestingly, there is not a monotonic relation. For example, for *BFS* and *FW* the grain-size for the training set is smaller than for the execution set because the work per task remains constant as the input size increases, but the parallelism increases; the larger grain-size serves the purpose of pruning more parallelism. Conversely, for *MM* the grain-size for the larger input-set is smaller than the grain-size of the training-set because the work per task increases linearly with N, and, consequently, less coarsening is needed to amortize scheduling overheads.

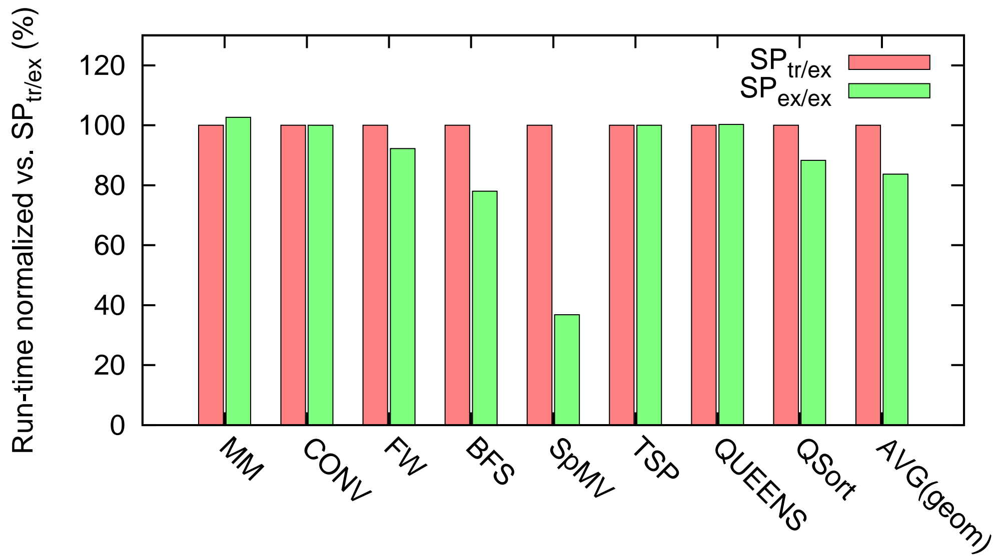


Figure 3.2: Performance Sensitivity of TBB's Manual Tuning

Figure 3.2 shows the normalized execution times on the large dataset of our implementation of TBB's simple-partitioner with the two grain-sizes, the one obtained using the training set ($SP_{tr/ex}$) and the one using the execution set ($SP_{ex/ex}$). The values used are the averages of a number of executions described hereafter. For *MM*, *CONV*, *FW*, *BFS*, *SpMV*, and *TSP* we ran each configuration only five times because the standard deviation was at most 0.06%. For *QUEENS*, we ran each configuration 20 times and the standard deviation was 1.8% for $SP_{tr/ex}$ and 0.27% for $SP_{ex/ex}$. The higher variability for $SP_{tr/ex}$ is explained by the fact that we picked a grain size for a recursively nested computation, trying to coarsen mid-tree (see

earlier discussion in Section 3.1) but failing to do so: some of the parallelism in the shallow recursive levels is effectively hidden from the scheduler (because of the depth-first execution order) resulting in more thefts of smaller tasks and in a larger variability of the execution. For *QSort*, we also noticed a high variability so we also executed each configuration 20 times and found the standard deviation to be 4.02% for $SP_{tr/ex}$ and 3.15% for $SP_{ex/ex}$.

The grain-size computed using the training set results in over 16% slower execution times than with the grain-size computed using the execution set. Of course it is unrealistic in general to perform the tuning on the execution datasets, as we argued. The benchmark that was affected the most was *SpMV* which took almost three times longer with the training grain-size. We see this performance degradation despite taking advantage of XMT's hardware scheduler. We believe that on traditional multicores the performance loss may be more severe, and we consider this result as a lower-bound on the harmful effects of manual coarsening of this nature.

One last thing to remember is that the execution context of a function plays a significant role in the grain-size needed. Generally, however, a function can be called from sequential and parallel contexts alike. In such cases, the unique grain-size computed using TBB's tuning method will miss the opportunity to adapt the coarsening to different contexts.

The main flaw of TBB's proposed tuning approach is that it tries to use static coarsening to tackle both goals of coarsening, amortizing the scheduling overheads per-task and pruning parallelism, inspite of the fact that pruning parallelism successfully depends on the input, the platform, and the calling context, which can wildly vary at run-time.

3.3.2 Sensitivity to picking the right cut-off for QUEENS

In this second experiment, we show that, even for a very straight-forward code such as QUEENS, coarsening can be challenging if the input size and the target platform are considered to be unknown variables.

We ran our experiments on an UltraSPARC-T2 running at 1.2GHz with 8 cores and 64 hardware threads, 4MB of L3 cache, and 32GB of DDR2. We used Intel's TBB library to parallelize QUEENS and used both the simple-partitioner and the auto-partitioner, TBB's default partitioner. Since the results for auto-partitioner were equal or slightly better than those for simple-partitioner, we only present results for the former.

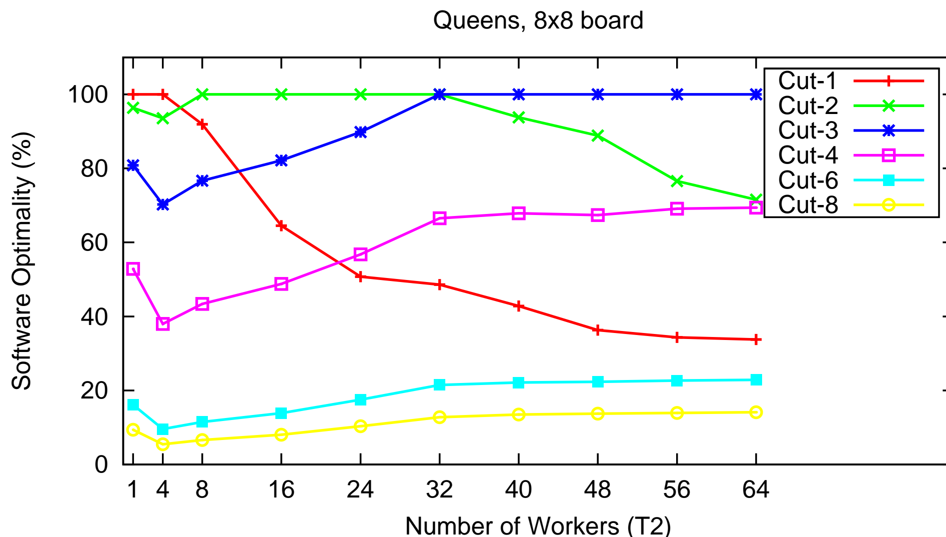


Figure 3.3: Sensitivity of Performance when varying the number of workers

We ran QUEENS on different input sizes $N \in D = \{4, 6, 8, 10, 12, 13\}$, with different numbers of workers $w \in W = \{1, 4, 8, 16, 24, 32, 40, 48, 56, 64\}$ and different cut-off depths $c \in C = \{1, 2, 3, 4, 5, 6, 8, 10, 12, 13\}$. For each point in that three dimensional space, we first found how many iterations of the computation we need to run so that the total run-time be above one second. We did that because we have found empirically that, for much shorter running times, events beyond our control

introduced substantial noise to our results, greatly increasing the variance of the measurements. Such events include context-switches, network requests, etc. For each point, we picked the average of ten executions, and we plot some of the results in Figures 3.3 and 3.4. We discuss the standard deviations for the measurements in these two figures after describing them.

Figure 3.3 shows the *software optimality* (shorthand for *software performance optimality ratio*) of each cut-off depth as a function of the number of workers used, on an 8 by 8 board. In section 3.4, we will define software optimality rigorously and compare it to the existing metric of efficiency. For our current example, software optimality is the ratio of the performance of an execution relative to the best performance achieved by any cutting depth:

$$SoftwareOptimality(N, c, w) = \frac{\min_{c \in C} Time(N, c, w)}{Time(N, c, w)}$$

The figure shows that, even for a fixed problem size, picking a cut-off depth without taking into account the number of workers may lead to sub-optimal performance. In our example, the cut-off depth of 1 is the best if we have few workers (1 to 4), the cut-off of 2 is best in the range of 8 to 32 workers, then the cut-off of 3 becomes optimal for the remaining range. Even if we only look at the cut-offs of 2 and 3, which are the best candidates, each gives sub-optimal performance by over 20% for some range of the number of workers.

As mentioned before, this is a hard problem to solve because it is not enough to know the number of workers that a machine has, but also how many of them are available at a particular point during execution. In other words, the number of available workers depends, at the very least, on the context in which a function was called. In multi-programmed environments, it also depends on the other processes currently contending for processing resources on the system. In short, it is not enough to model the number of workers as a constant parameter, but we should treat it as a dynamically fluctuating variable.

For the two best cut-offs (2 and 3), the standard deviation was below 0.42%. For the rest of the curves, it was below 1.3% except for the cut-off of 1 when using 32 workers which had an unexpectedly high standard deviation of 6.4%, possibly caused by an interference from an other process. That point is so far from competing with the cut-offs of 2 and 3, that it was not necessary to find the source of the variation.

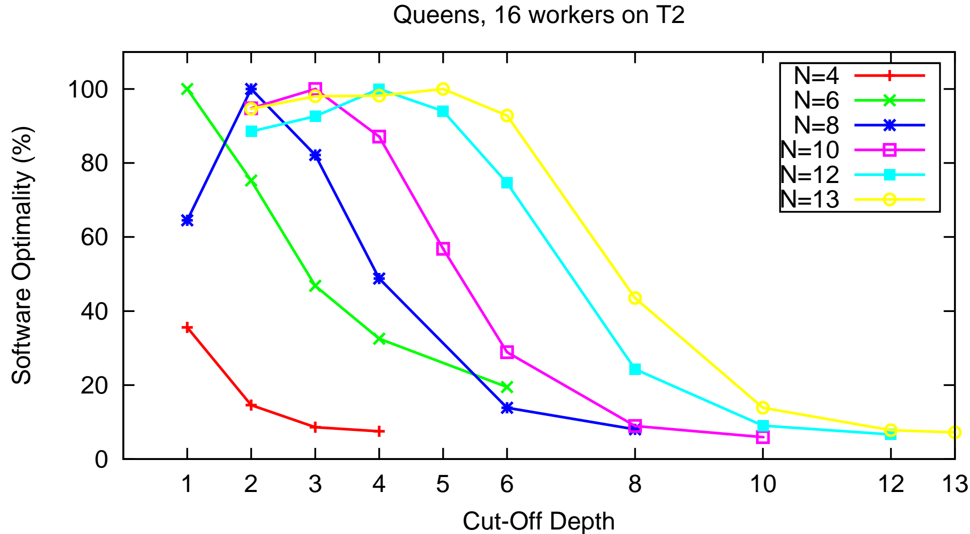


Figure 3.4: Sensitivity of Performance when varying the size of the input

Figure 3.4 shows the effect that varying the cut-off depth has on software optimality, for different input sizes, when executed on a constant number of workers; 16 in this case. Note that, for any input size k, a cut-off depth of $c > k$ has the same effect as no cut-off and is therefore omitted from the figure. Also, the curve for input size $N = 4$ reaches 100% software optimality for a cut-off depth of zero, i.e., sequential execution, which is not shown on the plot. The standard deviation for input sizes $N < 12$ is at most 0.48%; for $N = 12$ it is at most 1.8% and for $N = 13$ it is at most 2.53%. Increased variability is expected with larger inputs for an unbalanced computation such as *QUEENS*.

The thing to notice in Figure 3.4 is that the peak of each curve is at a different cut-off depth. Therefore, for each input size, a different cut-off depth gives the best performance. Moreover, for any depth one might pick, there are input sizes for

which the choice will be far from optimal.

This last observation raises the question of what methodology to use for comparing the quality of general-purpose parallel codes. The codes can be declarative or not, but should target an vast range of inputs, platforms, and calling contexts. A possible starting point is the type of experiment we just presented on QUEENS, but the result of that experiment is a large amount of multi-dimensional data which is hard to compare. The following section proposes an approach and two metrics for comparing the quality of such general-purpose parallel codes.

3.4 Evaluating the Quality of General Purpose Parallel Code: Proposed Framework

We propose two metrics, the *worst-case software optimality* and the *average software optimality*. The first one captures the minimum possible software optimality achievable, given a set of variables over which the programmer does not have control, such as the input and the target platform. The second metric captures the average software optimality in the same setup.

To formally define those metrics, as well as software optimality, we start by defining two groups of variables: the variables the programmer controls and the variables the *environment* controls. The variables under programmer control and their corresponding sets are the following:

$\mathbb{B}$: the set of implementations (including different algorithms) that we care to evaluate for the given problem.

$\mathbb{C}$: let C_b be the set of all possible (or reasonable) coarsenings for a given implementation $b \in \mathbb{B}$, then $\mathbb{C} = \bigcup_{b \in \mathbb{B}} C_b$.

$\mathbb{S}$: the set of all possible system configurations. This includes different user-level schedulers, dynamic memory allocators, garbage collectors, and generally all

the user-level “system code”.

This last set is important for this dissertation since we will show that lazy scheduling achieves better worst-case software optimality compared to auto-partitioner.

The variables the *environment* controls and their corresponding sets are listed below:

$\mathbb{I}$: the set of all inputs we care to evaluate. This variable is intended to capture performance portability across datasets (D).

$\mathbb{P}$: the set of all platforms we care to evaluate. This variable is intended to capture performance portability across different machines (M).

$\mathbb{W}$: let W_p be the set of all possible subsets of the computational resources of a platform $p \in \mathbb{P}$ that may be allocated for the execution of the code, then $\mathbb{W} = \bigcup_{p \in \mathbb{P}} W_p$. This variable is intended to capture the performance portability across calling contexts (C) and with multiprogramming by limiting the computational resources that are available for the execution of the code being evaluated.

One can add (or subtract variables) from the above framework to adapt it to their needs. The important thing is to distinguish the variables the programmer controls from the ones they do not.

Software Performance Optimality Ratio. Given the above state space, we define as *software performance optimality ratio*[2] of a code $b \in \mathbb{B}$ using a coarsening $c \in C_b$, with system code $s \in \mathbb{S}$, an input $i \in \mathbb{I}$, on a platform $p \in \mathbb{P}$, using a subset $w \in W_p$ of that platform as follows:

$$\mathit{SoftwareOptimality}(b, c, s; i, p, w) = \frac{\min_{b \in \mathbb{B}\ c \in C_b\ s \in \mathbb{S}} \mathit{Time}(b, c, s; i, p, w)}{\mathit{Time}(b, c, s; i, p, w)} \tag{3.1}$$

[2]We will use the term *software optimality* as a shorthand.

In the numerator, we take the minimum execution time over all combinations of implementations b (including sequential ones), coarsenings c, and system code configurations s, because the programmer has control over those parameters. *For clarity in the notation, we separate the variables the programmer controls from the ones they do not control using a semicolon.* The variables controlled by the environment (the input i, the platform p, and the subset w) are unknown to the programmer and affect the best achievable performance, so they are left out of the minimizing clause of the numerator.

The term *software* in the name of this metric underlines that the minimizing clause ranges over the parameters which the programmer controls, which are the software; this is also why the environment (i, p, w) is fixed on both sides of the equation. This is in contrast to the standard definition of efficiency (Equation 3.2 [36]), which intends to also capture hardware bottlenecks in a parallel execution, such as insufficient communication or memory bandwidth. The term *performance* underlines that we are interested in measuring performance (defined as the inverse of the running time), as opposed to efficiency, which normalizes performance by the number of workers and can be useful for comparing performance per watt. The term *optimality ratio* underlines that we are comparing the performance of a software configuration (b, c, s) to the optimal performance achievable by any software in our search-space, for a fixed environment configuration (i, p, w).Note that the definition of efficiency (Equation 3.2) also implies a fixed environment (i, p, w) but also a fixed "baseline", chosen to be a sequential code .

$$Efficiency = \frac{Speedup}{Number\ of\ Workers} \tag{3.2}$$

We find two issues with the definition of efficiency. First, it assumes that sequential code has 100% efficiency, which can result in efficiency values of more than 100% in the case of super-linear speedups, and second, it captures both software and hardware inefficiencies but without being able to point to the culprit. By con-

trast, our proposed software optimality metric only focuses on software performance, decoupling it from hardware bottlenecks. Thus, having both metrics, software optimality and efficiency, gives added value to the information conveyed by efficiency alone. Finally, note that, unlike efficiency, software optimality can never be greater than 1 (or 100%).

Worst-Case Software Optimality. We define *worst-case software optimality* for an implementation b, with coarsening c, using a system configuration s, to be the minimum software optimality over the sub-space covered by the variables that are not under the control of the programmer, i.e., in our particular set-up, the input i, the platform p, and its subset w. In other words, the worst-case software optimality is the global minimum of the software optimality function for fixed values of the programmer controlled variables b, c, s.

$$SwOpt_{WC}(b, c, s) = \min_{i \in \mathbb{I}\ p \in \mathbb{P}\ w \in W_p} SwOpt(b, c, s; i, p, w) \tag{3.3}$$

Average-Case Software Optimality. The average-case software optimality is defined similarly by replacing the minimum function by the geometric mean:

$$SwOpt_{Mean}(b, c, s) = \sqrt[\gamma]{\prod_{i \in \mathbb{I}\ p \in \mathbb{P}\ w \in W_p} SwOpt(b, c, s; i, p, w)}\ ,\ \gamma = |I| \cdot \sum_{p \in \mathbb{P}} |W_p| \tag{3.4}$$

We used the geometric mean instead of the arithmetic mean because software optimality is a ratio and not an absolute value.

Now, given these two metrics the programmer can try to find a triplet (b, c, s) that maximizes one or possibly both of them. E.g.,

$$(b, c, s)_{WC} = \underset{b \in \mathbb{B}\ c \in C_b\ s \in \mathbb{S}}{\arg\max}\ SoftwareOptimality_{WC}(b, c, s)$$

or

$$(b, c, s)_{Mean} = \underset{b \in \mathbb{B}\ c \in C_b\ s \in \mathbb{S}}{\arg\max}\ SoftwareOptimality_{Mean}(b, c, s)$$

One thing to note with this definition of the average software optimality is that all possible inputs, platforms and contexts (platform subsets) are weighed equally. For platform subsets, this can skew the results by giving much greater weight to subsets using approximately half of the resources $\binom{n}{n/2}$ than, say, to using the whole machine $\binom{n}{n} = 1$. This may be desirable if such platform subsets are considered more probable. Otherwise, appropriate weights can be introduced to remove the skew, or platform subsets can be divided in equivalence classes (e.g., class of all subsets containing half the workers) each counted once. Conversely, the worst-case software optimality does not hide such caveats since the minimum is taken and weights are not involved.

3.4.1 Discussion

Ideally, we would want to cover all possible inputs i, which are usually infinite, all possible implementations b, which can also be infinite, all possible coarsenings c, also potentially infinite, etc. Of course, explicitly covering such an infinite search space is impossible, so we must use good judgment in choosing which subset of this infinite search space to cover. How to do that is beyond the scope of this dissertation. For the worst-case software optimality, we could use standard techniques for finding minima in large state spaces, such as simulated annealing or hill climbing.

Given these realistic limitations in covering a substantial part of the state space, the measured worst-case software optimality has an advantage over the measured average software optimality; it represents an upper-bound of the actual worst-case software optimality. Of course, a lower-bound would have been preferable, but an upper-bound is still useful; it can unequivocally point out existing inefficient configurations, which represent corner cases the programmer did not consider while coding or coarsening. On the other hand, the measured average software optimality

could be smaller or greater than the "real" average software optimality, depending on the subspace covered and the weights used. For the measurement to hold useful information, one must carefully pick the subset of inputs, platforms and their configurations to explore and possibly pick appropriate weights for each of them. The quality of such choices are, of course, always up to interpretation, but reasonable compromises can probably be reached.

3.4.2 Using the Framework: An Example

In this section, we compute the worst-case software optimality for the QUEENS benchmark ($\mathbb{B} = \{QUEENS\}$) on the T2 platform ($\mathbb{P} = \{T2\}$) using TBB's auto-partitioner ($\mathbb{S} = \{AP\}$). We use inputs $\mathbb{I} = \{4, 6, 8, 10, 12, 13\}$ and platform subsets $W_{T2} = \{1, 4, 8, 16, 24, 32, 40, 48, 56, 64\}$ (the number indicates the number of worker threads, but not their assignment to hardware threads).

This example shows how the framework works, but it is not intended for programmers to use in the way demonstrated in this section. On the contrary, we hope to demonstrate how hard it would be for programmers to manually coarsen code while preserving performance portability, *even with the assistance of this framework*. In our view, the framework is a tool for compiler and runtime developers who want to test the performance portability of their contributions, as well as for testing the performance portability of proposed programming methodologies. For example, we will use the framework to argue that our proposed separation of coarsening goals is beneficial for performance portability.

The programmer's goal is to pick a coarsening that maximizes the worst-case software optimality. A programmer may start with cut-off depths $\mathbb{C} = \{1, 2, 3, 4, 5, 6, 8, 10, 12, 13\}$. The rightmost column in Table 3.3 shows the worst-case software optimality for each coarsening (cut-off depth), and the other columns show, for each input set, the "partial" minimum over all platform subsets $w \in W_{T2}$. The programmer may choose pick the coarsening that maximizes the worst-case software

$SwOpt_{WC(W_{T2})}(QUEENS, c, AP; i)$

	Size of N (Side of the Board)						
Depth	4	6	8	10	12	13	$SwOpt_{WC}(c)$
1	17.38	96.87	33.78	N/A	N/A	N/A	17.38
2	8.48	64.63	71.55	75.28	83.70	80.03	8.48
3	4.81	29.60	70.26	98.03	92.19	98.05	4.81
4	3.51	14.84	37.97	84.80	98.91	98.20	3.51
5	3.51	N/A	N/A	55.52	93.02	98.14	3.51
6	3.51	6.58	9.57	28.38	74.46	90.32	3.51
8	3.51	6.58	5.47	8.91	24.29	42.05	3.51
10	3.51	6.58	5.47	5.92	9.03	13.43	3.51
12	3.51	6.58	5.47	5.92	6.63	7.57	3.51
13	3.51	6.58	5.47	5.92	6.63	7.01	3.51

$\max_{c \in \mathbb{C}} SwOpt_{WC}(QUEENS, c, AP) = 17.38\%$, with $c = 1$.

Table 3.3: Worst-Case Software Optimality with constant cut-offs.

optimality, which is at a depth of 1. Note that some cells in the table are missing, but filling them can only reduce the measured worst-case software optimality. Since 17.38% is already disappointing, getting a more accurate upper bound is not very interesting.

An experienced programmer will realize that the cause of bad software optimality is the small input size ($N = 4$) that is not worth parallelizing. In response to that, they may change their coarsening approach and consider functions of the input size, such as serializing the last k recursive levels of the computation (cut-off-depth(N) $= N - k$) or coarsening after a few recursive levels (cut-off-depth(N) $= N/k$).

In this following experiment, we consider the following set of coarsening depths: $\mathbb{C} = \{$N-4, N-5, N-6, N/2, N/3$\}$ and repeat the computation of Table 3.3. Table 3.4

shows the results. For coarsenings of the form $N-k$ the software optimality decreases after some value of N, whereas for the N/k coarsenings, it increases. The best coarsening (of the ones explored) is at depth $N-6$ but only gets (at most) 50.17% worst-case software optimality. Also note that by looking at the $N-k$ rows we can deduce that it is probably not worth parallelizing QUEENS on T2 for $N < 6$.

$SwOpt_{WC(W_{T2})}(QUEENS, c, AP; i)$

	Size of N (Side of the Board)						
Depth	4	6	8	10	12	13	$SwOpt_{WC}(c)$
N-4	100.00	64.63	37.97	28.38	24.29	N/A	24.29
N-5	100.00	96.87	70.26	55.51	45.9	42.05	42.05
N-6	100.00	50.17	71.55	84.80	74.46	N/A	50.17
N/2	8.48	29.60	37.97	55.52	74.46	90.31	8.48
N/3	17.38	64.63	71.55	92.19	98.91	98.20	17.38
min(N/3,N-5)	100.00	96.87	71.55	92.19	98.91	98.20	71.55

$\max_{c \in \mathbb{C}} SwOpt_{WC}(QUEENS, c, AP) = 71.55\%$, with $c = \min(N/3, N-5)$.

Table 3.4: Worst-Case Software Optimality with cut-off functions.

An even more savvy programmer will try to combine amortizing scheduler overheads by using a $N-k$ function and pruning parallelism by using an N/k function. By looking at Table 3.4, one would pick $depth(N) = \min(N/3, N-5)$ to get the best of both amortizing and pruning and get a decent 71.55% worst-case software optimality.

This result already gives a small indication that tackling the two goals of coarsening separately helps preserve performance portability, but such a manual coarsening process is very tedious and all it achieved was a decent worst-case software optimality for a toy code, QUEENS. It is probably not something a general purpose programmer would want to do over a large code-base. One of the contributions of this dissertation is achieving such reasonable worst-case software optimality with

little or no involvement from the programmer. We do this by attacking the two goals of coarsening separately: by applying static coarsening to amortize overheads, and by using lazy scheduling to prune parallelism at run-time, when more information about the input and load is available. In the next section, we describe the static coarsening passes in the XMTC compiler tasked with amortizing the scheduling overheads.

3.5 Coarsening in the XMTC Compiler

In this section, we describe two compiler passes that try to determine how much parallelism is potentially profitable to expose to the scheduler. In other words, their goal is to coarsen very fine-grained tasks to amortize scheduling overheads. The first pass picks a grain-size for each spawn statement, and the second decides if a spawn statement should be converted into a sequential loop. Both passes use a cost estimation routine at their core to make these decisions, presented below.

3.5.1 Cost Estimation

The cost estimation is performed in the front-end of the compiler. Since we are only interested in detecting and coarsening very fine-grained tasks, we exclude tasks with loops and tasks that call functions. Furthermore, since we are only coarsening tasks at the leaves of the spawn-tree (see discussion in Section 3.1), we also exclude tasks that have nested parallelism. This makes cost estimation very easy, and as we will show, that is sufficient to achieve the goal in most cases. Extending this pass to include function calls (but not recursion), loops, and additional parallelism, is straight-forward with full program analysis.

The cost estimation pass is implemented in the front-end of the compiler because it is used by other front-end passes. It traverses the parse tree and aggregates costs using values for elemental operations from a table like Table 3.5. The unknown

Functional Unit	Cost (cycles)
Branch	3
ALU Operation	1
Integer Multiply	10
Integer Divide	20
Unknown Cost	1

Table 3.5: Example of Functional Unit Costs used for Task Cost Estimation.

cost is used for inlined assembly instructions. Generally the costs are optimistic, as we have found empirically that it is better to perform slightly more coarsening than necessary, rather than too little. For `if-then-else` statements, we assign the average of the estimated costs of the `true` and of the `false` branches. This assumes that the `true` branch will be taken 50% of the time. Better heuristics or profile data can help improve the accuracy of the cost estimator.

For a more accurate cost estimation, the pass should be done in the back-end of the compiler, after register allocation, so that the pass will know, for example, which values are read from registers and which from the cache, which may have significant latency on XMT. The costs should then feed back into the front-end passes, perhaps by compiling the code twice and keeping the cost estimation information between compilations.

3.5.2 Picking a Grain-Size

For each spawn statement, the XMTC compiler computes a grain-size parameter, unless one has been provided by the programmer. The cost estimation pass of Section 3.5.1 is called on the task code (the spawn block code). Unless the task code contains a loop, a function call, or a nested spawn statement, the grain-size is picked as the ratio of the minimum desired number of cycles for a task over the estimated cost of each task $\frac{min_{cycles}}{cost}$. The minimum desired number of cycles is picked so that

scheduling overheads are amortized. We picked a value of one thousand cycles to keep parallelism fine-grained but profitable. The value can be overridden using a compiler flag.

3.5.3 Serializing Spawn Statements (Parallel Loops)

Besides the scheduling overhead per task, there is the overhead of parallelizing what would otherwise be a sequential loop. This overhead includes outlining the body of the loop into a separate function that the scheduler can call (see Section 5.6.6). If the entire computation of such a parallel loop is short, it is preferable to execute it sequentially. If the number of tasks created by the parallel loop is known at compile time, the compiler will serialize or leave the spawn statement intact. Otherwise, it will create a sequential clone of the parallel loop and pick between the two at run-time, when the number of tasks is known.

This pass uses the cost estimation pass as a subroutine, and it will not serialize a spawn statement that contains nested parallelism, loops, or function calls. We used a threshold of ten thousand cycles below which a nested spawn is converted to a sequential loop. With careful tuning a better threshold may be found. Note that this threshold only applies for nested spawn statements because the parallelization costs for outer spawns are lower thanks to the XMT hardware support. There, a much smaller threshold would be needed but we have not looked into it.

The benefits of these optimizations are evaluated in the next chapter alongside our proposed lazy scheduler.

3.6 Conclusion and Future Directions

In this chapter, we characterized coarsening and identified the two goals it needs to achieve: amortizing scheduling overheads and pruning excessive parallelism. We presented experimental evidence that performing manual coarsening to

achieve both goals is hard and usually leads to loss of performance portability. To quantify that claim, we proposed a framework for evaluating what we defined as the *software performance optimality ratio* of general-purpose parallel code, and we defined two metrics, worst-case software optimality and average software optimality, to quantify the performance portability of said code. We argued that the goal of amortizing overheads needs to be met before tasks reach the scheduler and that static methods offer good solutions (experimental results are presented in the next chapter). Dynamic compilation can also provide solutions. On the other hand, the goal of pruning parallelism is harder because it depends on the input, the context in which the code is called, and the target platform. We consider dynamic methods, such as lazy scheduling presented in the next chapter, to be a better fit for this task. Finally, we presented two static coarsening optimizations that we implemented in the XMTC compiler to amortize the overheads of fine-grained tasks and to serialize parallel loops without enough work.

Our two static coarsening optimizations, while simple, yield good results in conjunction with lazy scheduling. They miss, however, one important class of parallel codes: the recursively parallel ones. These represent at least two important families of algorithms: divide-and-conquer and branch-and-bound. Automating the coarsening of such codes is an interesting topic for recent [34, 3] and future research. If the compiler can estimate the cost of a recursive function as a function of its inputs (e.g., depth), it will be able to coarsen such codes automatically by applying a depth based cut-off and by creating a sequential clone of the recursively parallel code.

Chapter 4

Lazy Scheduling

The lack of performance portability in the best existing schedulers (Eager Binary Splitting with simple-partitioner or auto-partitioner) is a serious issue for general-purpose parallel programming because, not only do we want code to run efficiently for different input sets and contexts, but we also want it to run faster on a variety of different existing and future parallel platforms with different numbers of cores. Ease-of-programming is also a crucial consideration: freeing the programmer from manually determining a fixed threshold for each do-all loop will shorten their development cycle and make them more productive. While AP does not *require* manual tuning, we will show that in cases with nested fine-grain parallelism its performance degrades and manually pruning parallelism is necessary for competitive performance.

In this chapter, we present the concept of *Lazy Scheduling* and three concrete variations of lazy scheduling based on work stealing. The concept of lazy scheduling is broader than work stealing, however, and it can be applied to other types of scheduling, but this is beyond the scope of this dissertation. Lazy Scheduling overcomes the drawbacks related to performance portability in simple-partitioner and auto-partitioner by not using any statically determined threshold to decide when to stop splitting a task descriptor and adding its fragments to the work-pool. Instead, it uses run-time conditions alone in making those decisions.

4.1 The two Insights of Lazy Scheduling

The first insight of lazy scheduling is that splitting a task descriptor and pushing it onto the shared work-pool (the local deque in work stealing) is likely

to be a wasted overhead if other workers are busy with other work. In such a situation, it is better for the worker to first execute some tasks from its current task descriptor without pushing work onto the work-pool, and then check the system load again to decide whether to split the remaining task descriptor. In this way, unnecessary splitting and work-pool transactions are avoided, but tasks are pushed on the work-pool when other workers are looking for work.

Directly implementing lazy scheduling to follow the above insight is not obvious because checking if other workers are hungry for work can be expensive. For example, maintaining global state such as a count of hungry workers does not scale without hardware support, and on the other hand, querying workers to see if they are hungry requires expensive remote accesses. Furthermore, the mechanism for checking if other workers are hungry must be light-weight, otherwise workers may stay hungry for a while before more tasks are pushed on the shared work-pool.

The second insight of lazy scheduling provides a light-weight heuristic for inferring the load of the system during run-time. It involves simply looking at the size of the shared work-pool, or parts thereof. In work stealing, for example, a worker looks at the size of its local deque, and if it is below a threshold, the worker pushes a task descriptor onto its deque. That is a good heuristic approximation for the system load because if the deque size is below a threshold (empty in our implementation), that is a strong indication that other workers were hungry and stole work from it. On the other hand, if the local deque is above the threshold, pushing tasks onto it is postponed, and the worker executes locally one or more tasks from the task descriptor. This results in dynamic load-based coarsening.

Unlike other work-pool transactions that have to be atomic, reading its size can be done in a racy way, as long as the error in the result is reasonable. For example, if the worker queries the size of its deque while a theft is performed, it is acceptable for the check to return any of the two values for the size. This is because a slightly stale value does not perturb the efficiency of the heuristic in practice.

Lazy scheduling creates a new logical state in which tasks may be, the *postponed* state. Postponed tasks are those that have become available for parallel execution, but the lazy scheduler has detected that the system is under load and has not yet placed them onto the shared work-pool; instead those tasks reside in the memory that is logically private to the worker that created them (e.g., its stack). A worker starts by working on its postponed tasks; then, in the case of lazy work stealing, it works on the tasks in its deque before trying to steal work.

4.2 Lazy Binary Splitting (Depth-First Lazy Work Stealing)

In this section, we describe the first implementation of lazy work stealing on XMT. We call it *depth-first* because it does not follow the breadth-first thefts order of work stealing, and while we were aware of the issue at the time, we did not see performance degradation on XMT. Depth-First Lazy Work Stealing was presented in [83] under the name of Lazy Binary Splitting (LBS).

Lazy Binary Splitting checks if the local deque is empty and only then splits the current task descriptor. Figure 4.1 shows how LBS works, including the deque-is-empty check for the reasons described above. Unlike deque transactions that often require expensive memory-fences, a deque-check does not and is therefore a very cheap operation.

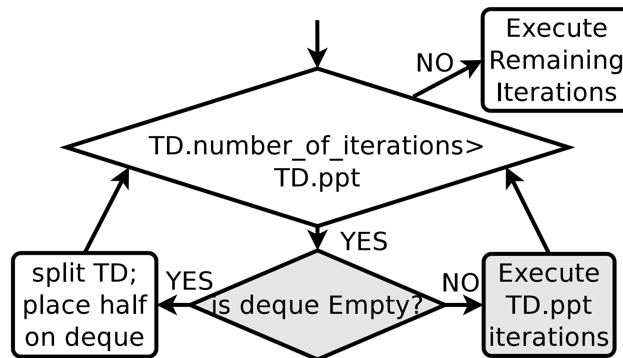


Figure 4.1: Processing a Task Descriptor with Lazy Binary Splitting (LBS)

Figure 4.1 shows an additional improvement in LBS – that it also stops splitting when the number of tasks in the task descriptor is equal to or below a statically-determined *profitable-parallelism-threshold* (ppt). This is present because creating very small amounts of parallel work is never profitable regardless of number of cores, datasets, or context, since the overheads of parallelism creation and synchronization will negate any gain from parallelism itself. The static coarsening compiler pass that picks the *ppt* was described in Section 3.5, and it was designed to amortize scheduling costs but not to prune parallelism. Because the *ppt* is independent of number of cores, datasets, or context, and because it only depends on the work per task and the implementation specific costs of creating parallelism, most times it can be easily determined by the compiler for each parallel loop without sacrificing performance portability. The performance portability of LBS comes from the deque-is-empty check, which ensures that enough but not too much parallelism is created for good load-balancing by adapting to run-time conditions. As explained in Section 3.5, when the parallel loop contains long-running tasks, such as nested parallelism, loops, or recursive calls, then LBS sets the profitable-parallelism-threshold to 1.

We now revisit the example we used to illustrate the shortcomings of serializing work stealing (Section 2.2.2.1) in order to show how LBS overcomes them. When lazy binary splitting is run, assuming processor A encounters a parallel loop with 16 tasks and a threshold (*ppt*) of 1, it creates a task descriptor with those 16 tasks and starts processing it (Figure 4.1): since the task descriptor has more than one tasks, it proceeds to check if the deque is empty; assuming it is, it splits the task descriptor and places half (tasks 9 to 16) on its deque. Then A starts working on iteration 1. Note that SP and AP would have continued splitting the task descriptor and pushing TDs with 4, 2, and 1 task before doing some actual work. In the meantime, processor B steals the task descriptor in A's deque and processes it: since B was a thief, its deque is empty, so B splits the TD and places half on its deque (iterations

13-16), and starts working on iteration 9. Then A finishes executing iteration 1, and processes its remaining TD (tasks 2-8): since A's deque is empty because of B's theft, A splits its TD, places half (5-8) on its deque and starts working on iteration 2. B finishes iteration 9, its deque is not empty so it continues with the remaining iterations in its TD (10-12) checking between each iteration execution if the deque is empty. Similarly A continues with its TD (3-4). When their TDs run out of iterations, A and B pop the TDs off their deques, split them, push half back on their deque, and work on their half. Actually, this sequence of pop-split-push operations is implemented as a single *pop-half* operation where half of the task descriptor is popped if it has more than *ppt* iterations. This is done to further reduce the number of deque transactions. For serializing work stealing the pop-take-an-iteration-push sequence is implemented similarly.

The example shows how LBS overcomes the serializing of task descriptor accesses by splitting them (like Eager Binary Splitting whether it be SP or AP), and also keeps the number of splits to a minimum by frequently checking the deque, making LBS more performance portable than Eager Binary Splitting. The next section provides a detailed comparison of the number of deque transactions and synchronizations for lazy binary splitting, eager binary splitting with simple-partitioner and auto-partitioner, and serializing work stealing (work-first). The comparison illustrates the benefits of LBS's run-time adaptivity to load conditions.

4.3 Analytical Comparison of Lazy Work Stealing with existing Work Stealing schedulers: A First approach

Unlike existing work stealing schedulers (e.g. SP, AP, SWS (i.e, work-first), and help-first work stealing), LBS is able to effectively combine iterations *at run-time* by postponing to push work while the local deque is not empty. This saves useless and expensive deque transactions which require memory-fences. It is easy

to appreciate the difference between SP, AP and LBS by analyzing the number of deque transactions and parent-child synchronizations (the main sources of overhead for work stealing) needed to schedule an N task parallel loop in the three scenarios described below. We call these three scenarios *worst*, *intermediate* and *best* because they require a decreasing number of deque transactions and synchronizations from all compared schedulers and especially LBS. Loosely speaking, an execution can be approximated as a combination of these three scenarios, which is why it is important to understand how the compared schedulers operate in these cases.

The results are summarized in Table 4.1. In the analysis below, we treat the *sst* and *ppt* thresholds (in SP and LBS respectively) as parameter *grain* and without loss of generality we assume that N is divisible by *grain* and both are powers of 2 to avoid cluttering the notation with floor and ceiling functions. We also assume that the *grain* parameter of the parallel loop is honored by SWS, and help-first. As we see, both transactions and synchronizations are linear in N for SWS, SP and help-first, but the situation for LBS is much different: the metrics go from linear in the worst case, to logarithmic in the intermediate case, to constant in the best case. AP's metrics go from linear in N in the worst case, to linear in P in the other two cases.

Worst Case: *When a worker encounters a do-all loop creating N iterations, and there are enough idle workers to immediately steal all TDs, effectively keeping all deques empty.* This happens, for example, when parallelism is first created by the original sequential thread, and it is barely enough to make all workers active ($N/grain \leq P$). In this case, SP and LBS behave identically: LBS always finds an empty deque because of the thefts, and keeps splitting and pushing TDs. Similarly the stolen TDs are split and stolen so eventually $N/grain$ TDs are created. That means that $N/grain$ parent-child synchronizations occur, one for each TD. Also $2(N/grain - 1)$ deque transactions happen: the factor of 2 accounts for the

	# Deque Transactions		
	Worst	Intermediate	Best
LBS($grain$)	$2\left(\frac{N}{grain}-1\right)$	$\log\frac{N}{grain}+1$	0
SP($grain$)	$2\left(\frac{N}{grain}-1\right)$	$\frac{3N}{2grain}-1$	$\frac{3N}{2grain}-1$
AP(K,V)	$2(N-1)$	$\frac{3K\cdot P}{2}-1$	$\frac{3K\cdot P}{2}-1$
SWS($grain$)	$2(\frac{N}{grain}-1)$	$\frac{N}{grain}$	$\frac{N}{grain}$
Help-First($grain$)	$2(\frac{N}{grain}-1)$	$2(\frac{N}{grain}-1)$	$2(\frac{N}{grain}-1)$

	# Synchronization Points		
	Worst	Intermediate	Best
LBS($grain$)	$\frac{N}{grain}$	$\log\frac{N}{grain}+1$	1
SP($grain$)	$\frac{N}{grain}$	$\frac{N}{grain}$	$\frac{N}{grain}$
AP(K,V)	N	$K\cdot P$	$K\cdot P$
SWS($grain$)	$\frac{N}{grain}$	$\frac{N}{grain}$	$\frac{N}{grain}$
Help-First($grain$)	$\frac{N}{grain}$	$\frac{N}{grain}$	$\frac{N}{grain}$

Table 4.1: Transaction and Synchronization Costs

push and steal transaction for every task descriptor, and the -1 accounts for the fact that one of the $N/grain$ task descriptors is never pushed on a deque, but is locally executed by the worker that created it. Similarly, for SWS and help-first, we have $N/grain$ synchronizations and $2(N/grain-1)$ deque transactions. For AP, assuming that $K\cdot P\leq N$, the task descriptor will be split into $K\cdot P$ chunks and incur $K\cdot P$ synchronizations and $2(K\cdot P-1)$ deque transactions, ignoring the possibility of further splitting induced by the thefts. Since thefts are generally rare (c.f., 4.7.2) , we are not giving AP a significant undeserved advantage by ignoring the theft-induced transactions and synchronization, so we chose to ignore them to simplify the analysis.

Intermediate Case: *When a worker encounters a parallel loop creating N iterations, the local deque is empty, but no thefts occur during its execution.* This can happen when a worker encounters a nested parallel loop while the outer parallelism was enough to feed all worker but not enough to fill the deques. This is very common in the XMT implementation as the outer parallelism is scheduled in hardware, and nested parallelism, which is scheduled using software, always finds the local deque to be empty. In the intermediate case, all N iterations will be executed on the worker creating them. For SP, and SWS, the difference of this intermediate case compared to the worst case is that some deque transactions can be combined, bringing their total number down. For SP $N/grain$ task descriptors will be created over the course of this execution, as in the worst case. One will never be pushed on the deque, but the rest will, resulting in $(N/grain - 1)$ pushes and $(N/grain - 1)$ pops. This number can be reduced if we use a pop-half transaction, which combines a pop and a subsequent push of half of the popped task descriptor. It is straight forward to show that the number of such pop-half transactions is equal to the number of nodes in a perfect binary tree[1] with $N/grain$ leaves, excluding the leaves, which represent the execution of an indivisible amount of work (*grain* tasks) and their parent nodes, which represent an indivisible task descriptor at the top of the deque which cannot benefit from the pop-half transaction. The number of the remaining nodes is $\frac{N}{2grain} - 1$, and the number of transactions becomes $\frac{3N}{2grain} - 1$. The number of synchronizations remains $N/grain$, as before. For AP, $K \cdot P$ task descriptors will be created, and following the same logic, the number of transactions will be $\frac{3K \cdot P}{2} - 1$, and the number of synchronizations will be $K \cdot P$. For SWS, the number of transactions is $N/grain$: one push of $N - grain$ iterations initially, followed by $N - 2$ *pop-grain* operations removing *grain* tasks each, and finally a pop of the remaining tasks. The number of synchronizations is also $N/grain$; one after every

[1] A binary tree that has all leaf nodes at the same depth and all internal nodes have exactly two children.

grain tasks. Help-first cannot benefit from the pop-grain transaction since it begins by creating $N/grain$ task descriptors that cannot be split further. For that reason, the number of transactions and synchronizations is the same in this case as in the worst case.

For LBS, the situation here is much different. Initially half the tasks ($N/2$) are pushed on the deque and the other half are executed, checking the size of the deque after every *grain* iterations but finding it full. Then, a pop-half operation reclaims half of the pushed tasks (i.e., $N/4$) which will be executed. Then, a pop-half will reclaim $N/8$ iterations, and so on, until the last $\frac{N}{2^k} = grain$ iterations are popped and executed. This amounts to $\log \frac{N}{grain} + 1$ transactions. The number of synchronizations is also $\log \frac{N}{grain} + 1$ because they happen before every pop-half, before the last pop, and at the very end.

Best Case: *When a worker encounters a parallel loop creating N tasks, no thefts occur and the deque is* **not** *empty.* This happens when nested parallelism is encountered, and the outer parallelism was sufficient to fuel all workers and deques, and it is particularly common for recursively nested parallelism.

For SP, AP, SWS, and help-first, nothing changes from the previous case, as these schedulers do not change their behavior based on the status of the deque. For LBS, things are very simple: no transactions occur and synchronization occurs only once, after all iterations have executed. We call this the *best case* because LBS incurs almost zero overhead in terms of deque transactions and synchronizations. In fact, even that single synchronization can be optimized away by detecting that none of the tasks were ever placed on the deque, but we have not implemented this optimization.

4.3.1 Deque Checks

So far we have focused on the overhead of deque transactions and synchronizations, but there is one more source of overheads in LBS: the checks to the local deque to decide whether to postpone pushing work or not. These checks are very light-weight and fast, but they are linear ($N/grain - 1$) in the number of tasks in all three cases presented above. Thus, for very fine-grained tasks they can become a significant source of overhead. Note that SP, AP, SWS, and help-first also perform deque checks to determine if pushing a task-descriptor will overflow the deque. In all three cases described above (best, intermediate, and worst), LBS, SP, SWS, and help-first perform $\mathrm{O}\big(N/grain\big)$ deque checks, while AP performs $\mathrm{O}\big(K \cdot P\big)$ checks. When iterations are very fine-grained, the linear overhead of these checks can become more important than the logarithmic or constant overhead of deque transactions and synchronizations of LBS. This motivates the need for having a profitable-parallelism-threshold for LBS, as will be described in the next section.

4.3.2 Role of the Profitable Parallelism Threshold (*ppt*)

As outlined earlier, the function of the profitable parallelism threshold (*ppt*) of LBS is to amortize scheduling costs by reducing the frequency of deque-checks, while the stop-splitting threshold (*sst*) of simple-partitioner focuses mainly on pruning parallelism to control the number of deque transactions and synchronizations by stopping the splitting. LBS achieves that goal by postponing pushing work onto the work-pool based on the deque size, the heuristic for gauging the system load. There is also a second source of overheads associated with the deque checks: the scheduler executes a task by calling its closure, and, to check the size of the deque, the execution must return to the scheduler code. So, for each deque-check, LBS also pays the overhead of a function call. Since these overheads are linear in the number of iterations, it is important to combine fine-grain iterations by means of

the profitable parallelism threshold (*ppt*). Remember that the *ppt*, also referred-to as *grain-size*, is picked by the compiler for each parallel loop, as described in Section 3.5.

Another thing to note from the analysis in Section 4.3 is that the *ppt* threshold (*grain*) in the intermediate and best cases plays a minimal role in controlling the number of transactions in LBS. The worst case, which is triggered by thefts, is rare enough, as backed up by our results in the rest of this chapter showing better performance for LBS, that it is fair to say that *ppt* is not the primary factor controlling the number of transactions and synchronizations in LBS. Conversely, *sst* is the only way these overheads are controlled in SP, SWS, and help-first. In AP, the only way to control the number of transactions and synchronizations is to also explicitly provide a *grain*, which supersedes AP's automatic coarsening. However, the *grain* parameter was not included in the analysis because AP is typically used without a grain-size – after all, this is its only advantage over SP.

4.4 Experimental Evaluation of Depth-First Lazy Work Stealing (LBS) on XMT

We ran our experiments on our 75MHz XMT FPGA prototype that is very similar to the one in [92]. The FPGA has 64 TCUs organized in 8 clusters, eight shared 32K L1 memory modules, and an 8x8 butterfly interconnection network connecting clusters to the L1 cache. There is one multiply/divide unit per cluster, each TCU has 4 prefetch buffers, and 32 integer registers. Floating point operations are not supported on that platform, so our benchmarks only perform integer computations.

We used the same benchmarks and datasets as the ones described in Tables 3.1 and 3.2 in Section 3.3.1.1, but we repeat these tables below for convenience (Tables 4.2 and 4.3. Table 4.3 also shows the automatically computed values for *ppt* (the

profitable parallelism threshold) that were automatically computed by our XMTC compiler.

Name	Description	DOP	Work/Task
MM	Dense Matrix Multiplication	N^2	N
CONV	N^2 image by M^2 filter convolution	N^2	M^2
FW	Floyd-Warshall all-pairs shortest path	N^2	1
QSort	Quicksort	$N/100$	QSort(100)
BFS	Breadth First Search	$O(\frac{\lvert E \rvert}{Diameter})$	O(1)
SpMV	Sparse Matrix by Vector Multiplication	$N = \#non\text{-}zero$	1
TSP	Travelling Salesperson Problem	$O(\frac{N!}{(N/2)!})$	$O(\frac{N}{2}!)$
QUEENS	placing N queens on an NxN board	$O(\frac{N!}{(N/2)!})$	$O(\frac{N}{2}!)$

Table 4.2: Summary of XMTC Benchmarks

	Training Set		Execution Set		AC
Name	**Size**	**grain**	**Size**	**grain**	**ppt**
MM	64x64	4	512x512	1	1
CONV	64^2 image, 16^2 filter	1	$1K^2$ image, 16^2 filter	1	1
FW	64 nodes	32	512 nodes	64	91
QSort	10K	16	1M	256	108
BFS	G(10K,200K)	16	G(10K,8M)	64	53
SpMV	30Kx100, 60K non-zero	4	80Kx5K, 40M non-zero	64	77
TSP	9 nodes	1	11 nodes	1	1
QUEENS	N=9	4	N=11	1	1

Table 4.3: Benchmarks, Datasets, and Grain-Sizes.

We timed five runs for each data point and used its average. Table 4.4 shows the standard deviation for the benchmakrs that did not have a trivial variability.

Note that for these experiments we did not focus on declarative benchmarks, since we wanted to compare lazy binary splitting to existing approaches on codes that they are supposed to support well. Later, we will show that on declarative codes the performance advantages of lazy scheduling are even greater.

We compared our LBS scheduler against several other schedulers shown in

Name	SWS	SP_1	AP_{xmt}	$AP_{default}$	$SP_{tr/ex}$	$SP_{ex/ex}$	LBS_1	LBS
TSP	0.02	0.02	0.01	0.01	0.02	0.02	0.53	0.50
QUEENS	0.37	0.27	0.30	0.33	1.80	1.22	1.16	1.16
QSort	1.77	1.85	3.12	2.62	2.32	3.11	5.02	3.88

Table 4.4: Standard Deviation(%) for Recursively Nested Benchmarks.

Table 4.5. All the compared approaches use the efficient hardware scheduler for outer parallelism provided by XMT and are only used to schedule the nested parallelism. We elaborate on the schedulers we have not described when we present the experimental results.

Name	**Description**
LBS	LBS (with *ppt* automatically determined by compiler)
LBS_1	LBS with *ppt=1 // Not Recommended; For comparison only.*
$AP_{default}$	AP with K=V=4 as in [76]
AP_{xmt}	AP with K=1, V=4, the best configuration for XMT
$SP_{tr/ex}$	SP with *sst* manually determined on training dataset and run on the execution dataset *// Realistic*
$SP_{ex/ex}$	SP with *sst* manually determined on execution dataset; then run on execution dataset. *// Unrealistic; For comparison only*
SP_1	SP with the default TBB threshold $sst = 1$ *// Not Recommended; For comparison only.*
SWS	Serializing Work Stealing (work-first) with $grain = 1$
SI	Serializing Inner parallelism

Table 4.5: Summary of Compared Configurations

LBS vs. AP: First, we compared LBS to $AP_{default}$, the configuration used in [76](i.e., K=V=4), and to AP_{xmt}(K=1, V=4), the optimal configuration for our benchmarks on XMT. We derived the values of K=1 and V=4 for AP_{xmt} by trying all nine configurations with $K, V \in \{1, 2, 4\}$ and picking the one that gave the best average performance on our benchmarks. We noticed that varying V for a given choice of K affected performance negligibly, so we picked V=4 for best adaptivity. While K=1 is low, it is acceptable on XMT because it will only be applied to nested parallel loops, as the outer parallel loop iterations are scheduled individually by

the hardware. Figure 4.2 shows that LBS is 16.2% faster than AP_{xmt} tuned for XMT, and 38.9% better than the default auto-partitioner configuration $AP_{default}$. AP falls behind on the benchmarks with very fine-grain parallelism (*FW*, *BFS* and *SpMV*). For those benchmarks, a manually determined *sst* would be required to further reduce splitting in AP. Since the values compared are normalized run-times (i.e., percentages) we use the geometric mean to calculate the average performance of each scheduler, not only in this comparison, but in the following ones as well. The standard deviations for *MM*, *CONV*, *FW*, *BFS*, and *SpMV* are below 0.12%.

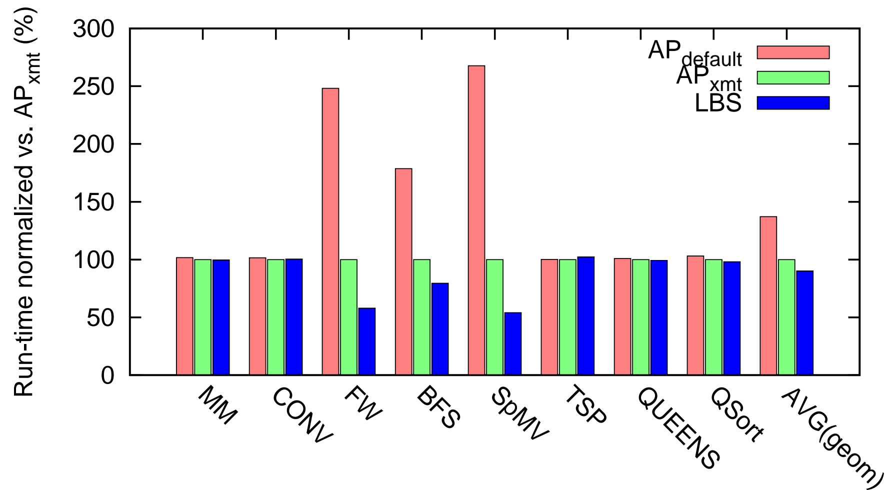


Figure 4.2: Comparing LBS to AP_{xmt} and $AP_{default}$

LBS vs. $\mathbf{SP}_{tr/ex}$ Next, we compared LBS and SP in their recommended configurations. For LBS, this is when *ppt* is determined by the compiler, while for SP, it is when *sst* is manually determined by the programmer using a training data set, and thereafter applied to run the execution dataset ($SP_{tr/ex}$). Figure 4.3 shows that on average LBS is 19.5% better and only falls behind on *TSP* (by 2.2%). For the other benchmarks, LBS is up to 65.7% better. This shows that LBS is not only easier to use since it needs no tuning, it also allows for more performance-portable code to any dataset (in this case *ex*) it encounters for the first time because, as we will see in our next comparison, the performance gap between LBS and SP diminishes when

SP is tuned on the execution dataset. The standard deviations of $\mathrm{SP}_{tr/ex}$ on *MM*, *CONV*, *FW*, *BFS*, and *SpMV* are at most 0.07%.

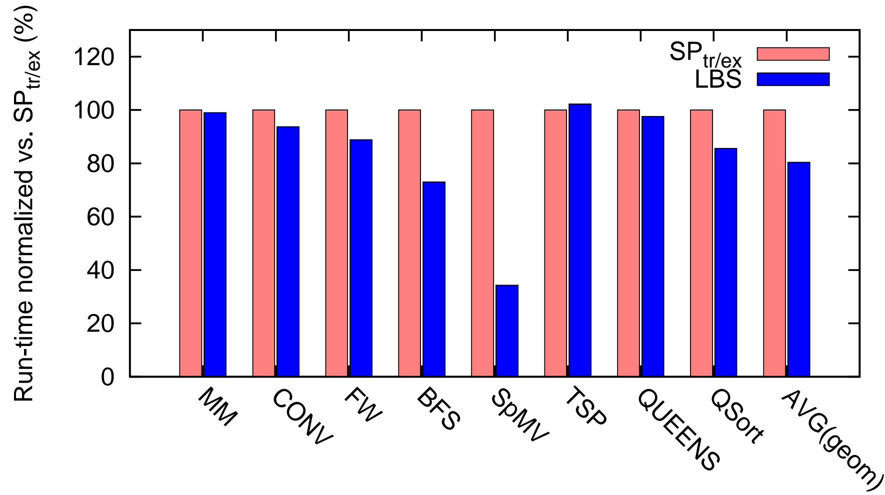


Figure 4.3: Comparing LBS to $\mathrm{SP}_{tr/ex}$

LBS vs. $\mathbf{SP}_{ex/ex}$ Next, we compared LBS to the hypothetical best case for SP (Figure 4.4): when SP is both tuned and run on the same execution dataset ($\mathrm{SP}_{ex/ex}$). This $\mathrm{SP}_{ex/ex}$ case is not realistic in general, since it makes no sense for the user of the program to tune SP for each new dataset, since typically datasets are different in each run in deployment. After all, multiple tuning runs are a waste, since after the first run of a dataset the program produces the required answer, and no further runs are needed.

This result is nevertheless presented to show that even in the ideal case for SP with idealized manual tuning on every new dataset, LBS (without tuning) still runs faster than $\mathrm{SP}_{ex/ex}$ by 3.8%, and falls behind only on *tsp* (by 2.2%). This means that even in rare cases when the datasets for an application have nearly identical characteristics, LBS is still a better choice – it is slightly faster, and a lot easier to use since no tuning is required. The greater gap between LBS and $\mathrm{SP}_{tr/ex}$ compared to $\mathrm{SP}_{ex/ex}$ reveals the superior portability of LBS to new datasets and run-time conditions. The standard deviations of $\mathrm{SP}_{ex/ex}$ on *MM*, *CONV*, *FW*, *BFS*,

and *SpMV* are at most 0.07%.

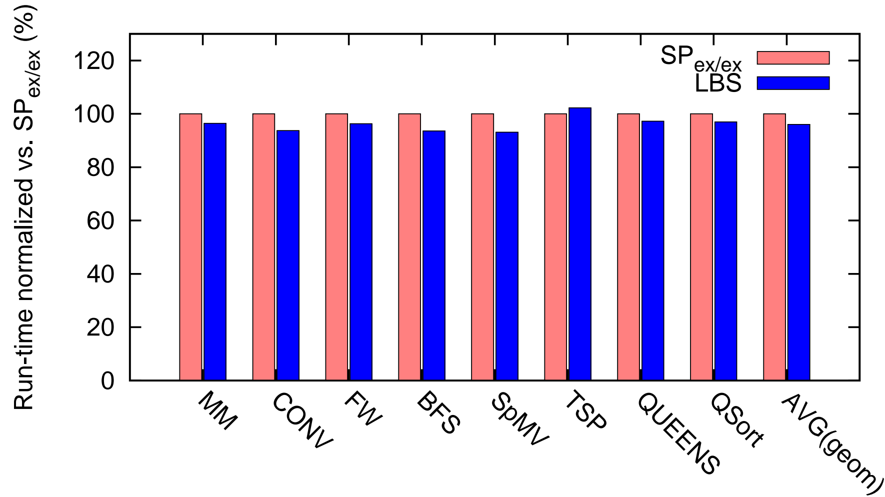


Figure 4.4: Comparing LBS to $\text{SP}_{ex/ex}$

LBS and LBS_1 vs. SP_1 and SWS Figure 4.5 first compares LBS, SP, and SWS in their best configurations that do not require any tuning, which are LBS, SP_1 and SWS, respectively. The goal was to compare performance at a constant user effort-level. SP is the only one of the three that needs tuning, and its best suggested configuration without tuning is SP_1, when the *sst* threshold is set to 1 (this is the default value in Intel's TBB when the user chooses not to do any tuning). SWS is the serialized work stealing scheme. Among these three (LBS, SP_1, and SWS), it is no surprise that LBS vastly outperforms the other two, by 56.7% and 54.7%, respectively, showing that, without tuning, LBS is the best choice.

We also present results for LBS_1 in Figure 4.5 to present an interesting (but not necessarily very meaningful) comparison between LBS_1 and SP_1. Neither has any compile-time restriction on splitting, and the comparison isolates the gain from the run-time adaptivity in LBS alone, which the figure shows is a sizable 47.2%. However, since both LBS and SP are run in sub-optimal configurations, we should not read too much into this result.

What is also interesting is that SP_1 is never better than SWS which confirms

our analysis in Table 4.1: when $sst = 1$ SP performs approximately 3/2 more transactions than SWS in the intermediate and best cases because the pop-one deque transaction benefits SWS more than the pop-half benefits SP. It is important to recognize, however, that XMT's hardware scheduling of outer parallel loops practically eliminates SWS's problem of serialized accesses to the task descriptor. Had we used SWS to schedule the outer parallelism as well, the performance might have been worse than that of SP_1.

The standard deviations for SP_1, SWS, and LBS_1 on *MM*, *CONV*, *FW*, *BFS*, and *SpMV* are below 0.12%.

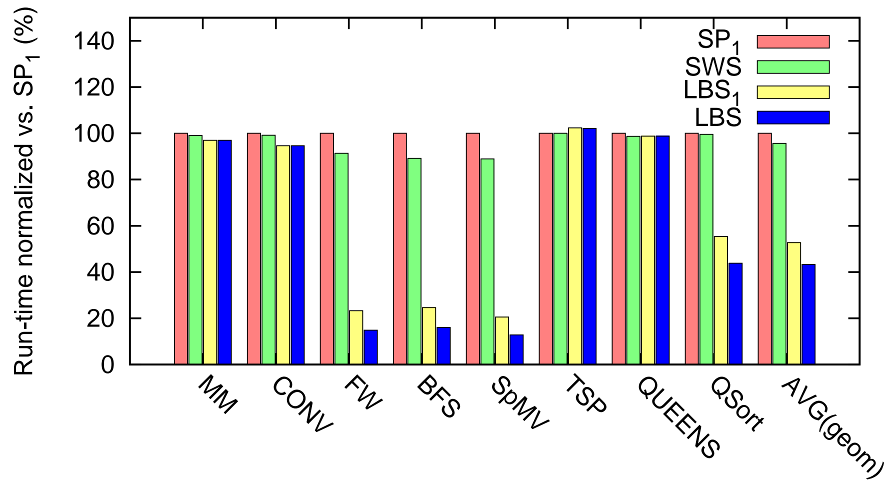


Figure 4.5: Comparing LBS and LBS_1 to SP_1 and SWS

LBS+ vs. SI Serializing Inner Parallelism (SI) simply serializes all inner parallelism. For example, nested parallel loops are converted to sequential ones. Since it is an easy way to provide some support for nested parallelism, it has been adopted by some OpenMP implementations, especially before version 3.0 when supporting nested parallelism became an explicit goal of OpenMP. We found that although overall LBS substantially outperforms SI, for three benchmarks (*FW*, *BFS* and *SpMV*) LBS falls behind. The worst was *FW* where LBS was performing much worse than SI because the inner parallelism was extremely fine-grained and regular and the over-

head of even creating task-descriptors and running the software scheduler (LBS) was excessive. But, because the bounds of the parallel loop were known at compile time, the compiler decided that it is not profitable to parallelize it and chose to serialize it, as described in Section 3.5. LBS+ includes this additional optimization.

For *BFS* and *SpMV*, the situation is more complex because the bounds of the inner parallel loops are only known at run-time just before they are executed. Our compiler injects a check in the code just before the inner parallel loop to decide whether to run a serialized clone or the original parallel loop. This decision is based on the number of iterations of the parallel loop and on a statically determined estimate of the amount of computation (in cycles) each iteration will perform. We run the parallel version when the total estimated computation of the parallel loop exceeds ten thousand cycles. We call this configuration LBS+ to distinguish it from the LBS configuration we used so far, where the optimization of static and dynamic serialization of inner parallelism was turned off to make the comparison to the other approaches fairer. Note that LBS+ and LBS are different only for *FW*, *BFS* and *SpMV*.

Figure 4.6 shows that on average LBS+ outperforms SI by 54.2%, doing much better than SI on code without enough outer parallelism (*TSP*, *QUEENS*, *QSort*) and doing as well as SI on code with longer iterations (*MM*, *CONV*), which shows that the additional overhead of scheduling inner parallelism is negligible in those codes. LBS+ still falls behind on *BFS* and *SpMV* because the injected check to decide whether to run the parallel or serialized version of the parallel loop can take several tens of cycles (because it needs to access memory locations), which might be a significant enough percentage of the computation of the inner do-all. Overall, however, having the checks to pick between a serialized or the original parallel clone of a parallel loop is very beneficial.

The standard deviation for LBS+ for the three benchmarks it affected (*FW*, *BFS*, and *SpMV*) was below 0.08%. For SI, the standard deviation for all bench-

marks was at most 0.06%: since nested parallelism and software scheduling was not involved, the amount of randomness and variability was greatly reduced.

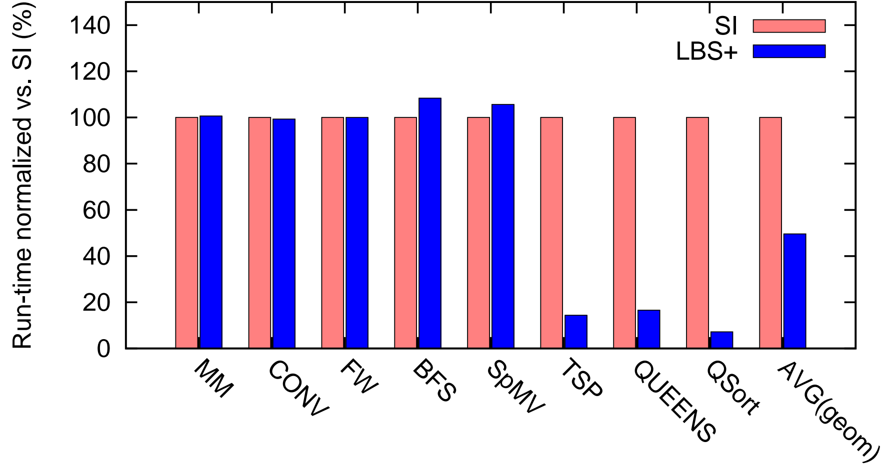


Figure 4.6: Comparing LBS+ to SI

4.4.1 Scalability and Speedups

Table 4.6 shows speedups with LBS+ on all 64 TCUs (parallel cores) compared to running the same parallel program with LBS+ on one TCU of the XMT prototype. The average speedup of 62.3x shows that LBS+ scales well to a significant number of cores. Some of the speedups are super-linear, which is explained by complex cache behavior causing more cache misses when only one TCU is active.

MM	**CONV**	**FW**	**BFS**	**SpMV**	**TSP**	**QUEENS**	**QSort**
70.5	67.2	54.7	63.2	60.7	67.5	62.5	52.1

Average Speedup (arithmetic): 62.3

Table 4.6: Speedups of LBS+ vs. Parallel Program on 1 TCU

We also present speedups of our programs compared to an optimized sequential version that runs on the powerful MTCU of XMT in Table 4.7. While the

previous numbers reveal that LBS scales well to many cores on the XMT architecture, these numbers show the attainable performance on XMT. It is fairer to judge performance using these numbers for two reasons: TCUs are much simpler and lightweight compared to the MTCU, which is a powerful sequential core that should be our baseline, but, also, a sequential program is usually simpler and requires less computation than its parallel counterpart. For example, in *TSP* the parallel version uses dynamic memory allocation to build possible solutions in parallel, whereas the serial version can use a single, statically declared array, which is why *TSP* has a smaller speedup than other benchmarks.

MM	**CONV**	**FW**	**BFS**	**SpMV**	**TSP**	**QUEENS**	**QSort**
63.9	28.2	37.6	12.8	26.0	11.1	20.1	6.9

Average Speedup (arithmetic): 25.8

Table 4.7: Speedups of LBS+ vs. Serial Program on MTCU

Overall, the average speedup of 25.8x is impressive given that the serial code is more efficient, the MTCU is much more powerful than the TCUs, and that several of our benchmarks are irregular and hard to parallelize. Unlike on XMT, for many other platforms and compilers, irregular benchmarks yield little or no parallel speedup.

4.5 Scalability Issues of Depth-First Lazy Work Stealing

The good theoretical bounds of work stealing rely on the scheduling order: execution proceeds in a ***depth-first*** order by treating the local deque as a stack, and thefts follow a ***breadth-first*** order by stealing the oldest task descriptor on a deque. Lazy Binary Splitting violates the breadth-first thefts order because, when it finds the local deque to be empty, it pushes work from the task descriptor being processed, instead of the oldest (i.e., outermost) postponed task descriptor owned

by that worker. Often, deeply nested tasks contain less work in their computation sub-tree than shallower tasks, and LBS ends up making smaller chunks of work available to hungry workers by pushing the innermost postponed tasks instead of the oldest postponed task. This, in turn, leads to more frequent thefts of smaller amounts of work, and thus more overheads.

On multicores, thefts are more expensive than on XMT because their memory hierarchy includes private caches. First, stealing a task descriptor involves acquiring exclusive write permission to a cache-line that is typically owned by the victim worker. Furthermore, the activation frame for that task typically also resides in the private cache of the victim worker. The theft can be thought of as a light-weight context-switch where parts of the private cache of the victim worker are transferred to the thief. On XMT, because it does not have private caches, a theft only involves stealing a task descriptor, an operation that is almost as cheap as popping a task descriptor locally.

The evaluation of LBS on XMT did not reveal scalability problems, but, as mentioned, thefts on XMT are not as expensive as on multicores, and the selected benchmarks were not well suited to trigger a bad behavior from LBS. For the most part, the benchmarks had two nested parallel loops, of which one was scheduled by the hardware, and the other by LBS, in which case the breadth-first theft order was honored. The three recursively nested benchmarks (*TSP*, *QUEENS*, *QSort*) were not declarative, but included a parallelism cut-off after a certain depth. Therefore, LBS would only push shallow tasks on the deque, as deeper ones were serialized, thus preventing the number of thefts from increasing significantly and, with it, the scheduling overhead.

Bergstrom et al.[13] implemented *Lazy Tree Splitting* (LTS), a variant of LBS for functional programming languages that use tree representations for arrays. They also note the potential for scaling issues of LBS and their derived LTS scheduler, but do not find indications of that experimentally. We attribute the lack of negative

results to the fact that they only had a 16 core machine and that their implementation of arrays induced some parallelism coarsening, not allowing parallelism to fully expand. In our experiments, we also found that LBS scaled well on non-declarative codes.

To show that LBS has scaling issues on traditional multicores, we implemented it in Intel's Threading Building Blocks library (TBB v3.0). TBB implements work stealing and provides the programmer with an API that implements parallel loops, sum-like reducers, and other operations. We chose TBB because parallel TBB code achieved good speedups versus serial implementations, indicating that TBB is implemented efficiently, and because TBB supports various target platforms, which allowed us to run experiments on a variety of machines. Cilk does not support parallel loops, so it was not a candidate.

Algorithm 4.1 Queens declarative pseudocode. $depth \in [1, N]$

```
1:  procedure QUEENS(N, partial_sol, depth)
2:      for all i ∈ [1, N] do
3:          if OK_TO_ADD(i, partial_sol, depth) then
4:              append i to partial_sol
5:              if depth < N then                                  ▷ Recursion
6:                  QUEENS(N, partial_sol, depth + 1)
7:              else                                          ▷ Found a Solution
8:                  atomic(global_sol_count+ = 1)
9:              end if
10:         end if
11:     end for
12: end procedure
```

To demonstrate the lack of scalability of LBS we use *QUEENS* (with N=14) for its recursive nested parallelism, but without parallelism cut-off (same as Algorithm 3.3 but without lines 6,7 ,8, and 10), to intensify the repercussions of choosing the innermost task, as shown in Algorithm 4.1 . Since our goal is to set the foundations of efficient support for declarative parallel programming, it is important to ensure good scalability in the absence of manual coarsening.

We used three commercial multicores for our evaluation, summarized in Ta-

Name	i7	Xeon	T2
CPU	i7 CPU 920	4 Intel Xeon E7450	UltraSPARC-T2
Clock	2.67GHz	2.4GHz	1.2GHz
Cores	4	24	8
Threads	8	24	64
L3 cache	8MB	4×12MB	4MB
RAM	6GB DDR3	48GB DDR2	32GB DDR2
kernel	linux 2.6.35	linux 2.2.26	Solaris 5.10
g++	4.4.5	4.1.2	3.4.3
libc	2.12	2.5	N/A

Table 4.8: Platform Descriptions

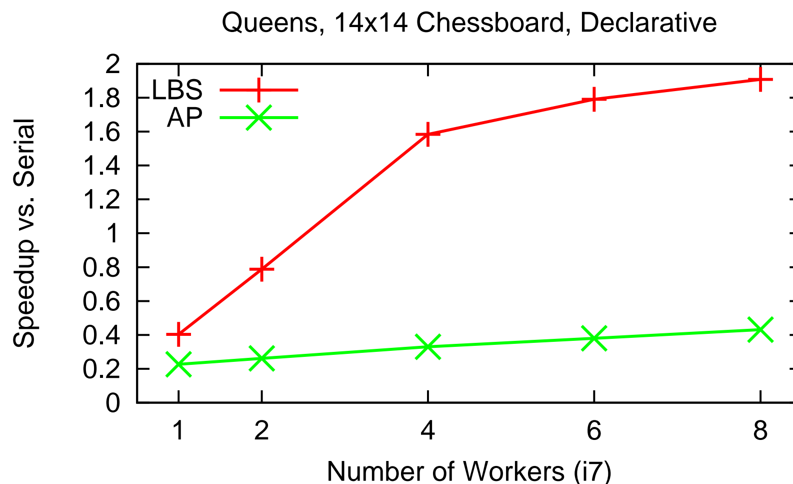


Figure 4.7: Scaling of schedulers on i7 (Queens)

ble 4.8. The three machines are very different and include a multicore desktop (i7), an SMP multicore (Xeon), and a Niagara2 multithreaded multicore (T2). Figures 4.7, 4.8, and 4.9 show the performance scaling of LBS on QUEENS on these three machines. Each data point is computed as the average of 10 executions. The standard deviations are presented in Tables 4.9, 4.10, and 4.11. The performance of auto-partitioner (AP), TBB's default scheduler, is also shown for reference. TBB's simple-partitioner is not shown because auto-partitioner outperformed it.

# Workers	BF-LS	DF2-LS	LBS	AP
1	0.99	0.98	1.92	0.31
2	0.37	0.44	1.64	1.27
4	2.48	1.58	1.26	1.82
6	0.37	0.45	0.89	0.65
8	0.49	1.19	2.44	0.67

Table 4.9: Standard Deviation(%) for i7 (Figures 4.7 and 4.12.)

# Workers	BF-LS	DF2-LS	LBS	AP
1	1.40	0.75	0.71	0.97
2	0.73	0.65	0.47	1.57
4	0.28	0.33	0.27	0.67
6	0.26	0.15	0.70	0.95
8	0.15	0.48	1.58	0.43
12	1.41	0.21	4.01	0.86
16	1.04	5.07	2.99	0.84
18	0.26	1.11	1.62	0.25
22	0.53	0.60	1.82	0.35
24	1.48	0.43	3.39	0.62

Table 4.10: Standard Deviation(%) for Xeon (Figures 4.8 and 4.11.)

# Workers	BF-LS	DF2-LS	LBS	AP
1	0.36	0.28	0.36	1.14
2	0.22	0.12	0.25	0.35
4	0.08	0.10	0.12	0.33
6	0.08	0.07	0.38	0.21
8	0.07	0.13	2.28	0.20
12	0.18	0.10	7.69	0.10
16	0.12	0.12	12.90	0.15
18	0.14	0.08	9.63	0.36
22	0.29	0.22	7.56	0.36
24	0.82	0.35	2.28	0.65
32	1.64	0.92	8.19	1.59
40	0.80	0.63	3.66	0.41
48	0.87	0.99	4.26	0.58
56	1.00	2.71	5.96	0.42
64	0.27	2.00	6.82	1.17

Table 4.11: Standard Deviation(%) for T2 (Figures 4.9 and 4.10.)

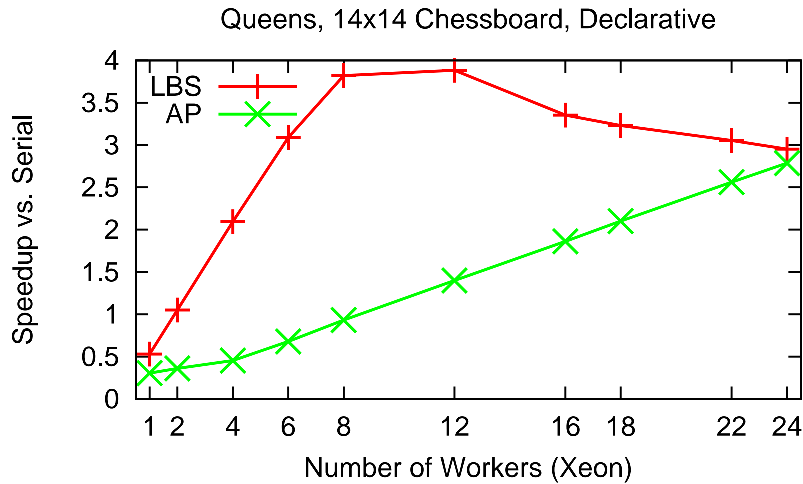


Figure 4.8: Scaling of schedulers on Xeon (Queens)

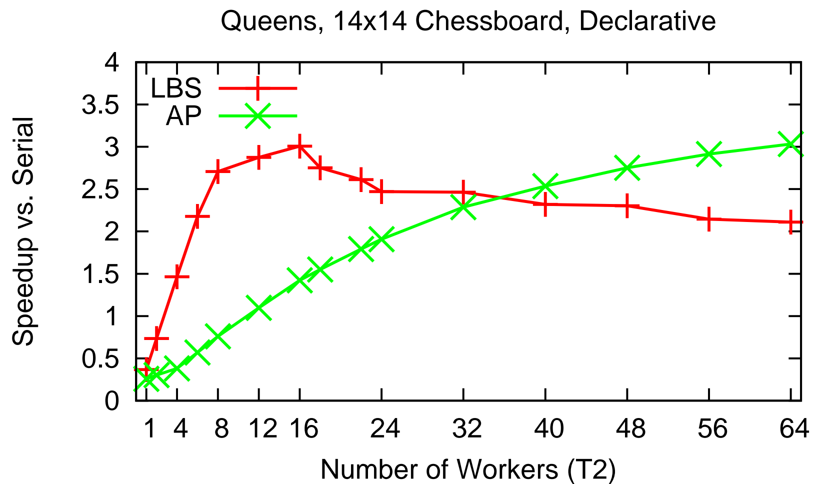


Figure 4.9: Scaling of schedulers on T2 (Queens)

On the small i7 machine (Figure 4.7), LBS scales well and greatly outperforms AP. The sub-linear performance scaling of LBS is attributed to the fact that when we initiate workers (threads) than the number of cores (4), hyper-threading kicks in and the workers compete for shared core resources.

On the two larger machines, Xeon (Figure 4.8) and T2 (Fig 4.9), LBS fails to scale. It scales well up to 8 workers, which shows the promise of lazy scheduling, but then flattens out and even decreases in performance as more workers are used.

On T2 in particular, the performance falls below the performance of AP when using many workers. LBS's performance degrades because workers push their innermost postponed tasks at the moment when they discover their deque to be below the threshold, which can contain an exponentially smaller amount of work than their outermost postponed tasks. This greatly increases the number of thefts, the most expensive scheduling operation, and prevents LBS from scaling to larger numbers of workers. In Section 4.7.1, we will show that we can fix the scaling problem of LBS and allow it to scale to larger machines. We will also count the thefts for the compared approaches and show that LBS incurs by far the most thefts. In the next section we present two ways to make lazy scheduling scale on multicores by reducing the frequency of thefts it incurs.

4.6 Lazy Scheduling for Declarative Code

We describe two approaches for solving the scalability issues of LBS with declarative code. The first, more robust but a bit harder to implement, involves pushing the outermost postponed tasks instead of the innermost ones. The second, less robust but trivial to implement, actually achieves comparable performance to the first one on our set of benchmarks and may be a reasonable alternative for mid-size machines (between 10 and 60 workers). It involves increasing the deque size threshold of LBS from 1 to 2. This simple change has deeper effects on the scheduling algorithm than is immediately apparent; these are described in Section 4.6.2.

4.6.1 Breadth-First Lazy Scheduling (BF-LS)

In Eager Scheduling, a worker always pushes tasks on its deque as soon as it encounters them. Conversely, in Lazy Scheduling, if the system has enough parallelism, a worker postpones pushing tasks on its deque. Later during the execution, the worker can discover that other workers are hungry and decide to push work onto

the shared work-pool (e.g., its deque). At that point, the worker may have many pending task descriptors to choose from if the code has nested parallelism.

From an implementation standpoint, the simplest solution is to push the innermost postponed task – the task descriptor being processed at the time of the decision to push work onto the work-pool. We call this approach *Depth-First Lazy Scheduling* (DF-LS), and it is the approach of LBS [83] and LTS [13]. While, in our experience, LBS works well on XMT for the benchmarks we tried (see Sections 4.4 and 4.8), using it on commercial multicores is a bad idea because *(1)* it pushes deeply nested tasks which are likely to contain less work than shallower tasks, leading to more thefts and deque transactions (pushes & pops), which we are trying to reduce by lazy scheduling in the first place, and *(2)* the theoretical bounds on work stealing rely on the principle of *breadth-first thefts*, which DF-LS violates. In fact, in the previous section we have shown experimentally that LBS (i.e., DF-LS) fails to scale to large numbers of workers on multicores for a recursively nested declarative parallel code. On the other hand, the depth-first approach may reduce the memory footprint in practice, although it is unlikely that a better theoretical bound can be proven.

The dual approach to DF-LS is to push on the deque the oldest postponed task, which also has the shallowest nesting depth. This approach honors the principle of breadth-first thefts, so we call it *Breadth-First Lazy Scheduling* (BF-LS). This approach is a bit trickier to implement because we must keep track of the postponed tasks and be able to push them on the deque, for which we use an additional list data structure. Because of this added bookkeeping, BF-LS has a slightly higher scheduling overhead per-task.

A third approach could push postponed tasks that are somewhere between the outermost and the innermost ones. From an implementation standpoint, that would incur the same (or more) bookkeeping overhead as BF-LS. The only apparent advantage of this approach is that it could reduce the memory footprint without

dramatically increasing the number of thefts, however, it would still violate the principle of breadth-first thefts.

Algorithm 4.2 BF-LS Scheduling of a TD representing a parallel loop

1: **procedure** PROCESSTASKDESCRIPTOR(td)
2: $totalExec \leftarrow 0$ ▷ Number of tasks executed
3: $savedTail \leftarrow worker.tail$ ▷ local var used to restore tail
4: Enqueue td at $worker.tail$ ▷ tail of list of postponed tasks
5: **while** $td.nrt > td.grain$ **do** ▷ number of tasks > grain
6: **if** $worker.dequeSize < THRESHOLD$ **then**
7: $tdToPush \leftarrow worker.head$
8: split or push $tdToPush$ onto $worker.deque$
9: **else** ▷ Execute sequentially $td.grain$ tasks
10: $id \leftarrow td.id$; $td.id \leftarrow td.id + td.grain$
11: $td.nrt \leftarrow td.nrt - td.grain$
12: $td.func(id, td.grain, td.args)$ ▷ Execute $grain$ tasks
13: $totalExec \leftarrow totalExec + td.grain$
14: **end if**
15: **end while**
16: **if** $worker.head \neq worker.tail$ **then** ▷ restore $tail$
17: restore $worker.tail$ using $savedTail$
18: **else** ▷ $savedTail$ was consumed
19: $worker.head \leftarrow worker.tail \leftarrow NULL$
20: **end if**
21: **if** $td.nrt > 0$ **then** ▷ execute any remaining tasks
22: $td.func(td.id, td.nrt, td.args)$
23: $totalExec \leftarrow totalExec + td.nrt$
24: **end if**
25: **return** $totalExec$ ▷ Used to decr. continuation's pending tasks
26: **end procedure**

Algorithm 4.2 presents the implementation of BF-LS. For simplicity we assumed that task descriptors represent 1D iteration ranges. We increment each postponed TD with a pointer to the next postponed task descriptor and create a linked list of postponed TDs per worker, with the oldest (outermost) TD at the head of the list and the newest (innermost) at the tail. Alternatively, the task descriptor can be kept unmodified but encapsulated in a wrapper structure that contains the pointer to the next postponed task descriptor. In this case, each *worker* maintains pointers to the head and tail of its list, in addition to maintaining its deque. The head will be used to push (part of) the oldest TD on the deque, and the tail will be

used to easily add TDs to the end of the list. TDs also need to be removed from the end of the list as they are consumed (by the depth-first execution). This is done on lines 3 and 16-20 of Algorithm 4.2.

The access pattern of this list of postponed task descriptors is very similar to that of a deque. New postponed task-descriptors are added at one end and consumed from the same end; when work is needed on the deque, work is removed from the other end of the list. Keeping this list has the following advantages compared to just pushing all the work on the deque: *(1)* the list is private to the worker and does not need any synchronization, unlike the deque which is shared; *(2)* a task descriptor is not recursively split to be added to the postponed list, but it is added as-is and modified in-place when it is being consumed or split to feed the deque; *(3)* the list elements are allocated on the stack, so there is no need for additional expensive dynamic memory allocation.

One subtle point of this algorithm is that, on lines 10 and 11, the worker must remove the tasks from the td before executing them. This is because an executed task may create additional tasks td', at which point the worker may push the rest of td on its deque. If the tasks were removed after they were executed, td could be added to the deque including the tasks being executed, leading to a double execution of those tasks, which is generally not correct if the task has side-effects. Relaxing the requirement of unique execution can lead to less strict synchronization requirements and improve performance, even though some work is duplicated [61, 66]. In our case, however, duplicate execution violates correctness.

Another thing to notice is that on lines 16-20 of Algorithm 4.2, we must check if the list of postponed tasks has been consumed (by line 8) in order to correctly remove the tail of the list, which we are about to consume (lines 21-24).

4.6.2 DF-LS with a threshold of 2 (DF2-LS)

Simply increasing the deque size threshold of LBS from 1 to 2 can drastically improve its performance scaling. We call this algorithm DF2-LS to distinguish it from DF-LS (i.e., LBS) which has a threshold of 1. The fundamental difference between the two is most noticeable when the machine is starving for parallel work, i.e., when many workers are trying to steal and most deques are empty, as occurs when switching from sequential to parallel execution for example. During that initial period, a few workers have tasks to push onto their empty deques. If the deque threshold is 1, the worker will push some tasks onto its deque, immediately check its size, and find it equal to the threshold, because thieves have not had time to steal the work. Consequently, the worker will falsely conclude that other workers are not hungry, and will start executing a task. If nested parallelism is encountered, the worker will discover that its deque is empty and push some of the inner tasks onto its deque, instead of the outer ones. Conversely, with a threshold of 2, the worker pushes some tasks onto its empty deque, then pushes some more of its tasks. This second round of pushing tasks gives thieves enough time to steal the tasks the worker pushed during the first round. The worker will subsequently keep pushing outer tasks until thefts become less frequent and the system is no longer starving for work. Therefore, having a threshold of 2 effectively creates a ***delay*** between pushing work onto the deque and checking to see if thefts have occurred, making the heuristic of checking the size of the deque a more accurate indicator of the system load.

We also experimented with adding an artificial delay between pushing work on a deque and the subsequent size check of that deque, instead of increasing the threshold to 2. We called `usleep` with arguments ranging from 1 to 25, but always noticed a performance degradation compared to LBS. We did not try to implement the artificial delay as a shorter busy wait (a loop of, say, 100 iterations that do nothing), however, because we do not believe that wasting power to simply wait is a good strategy, especially since power is already one of the factors that limit

performance.

Despite the good scalability results that we will present in Section 4.7.1, DF2-LS remains a depth-first approach, with the same problem of pushing the innermost tasks and incurring an increased number of thefts. Moreover, in the absence of nested parallelism, the higher deque threshold of DF2-LS causes more deque transactions (pushes and pops), without any additional benefit since all tasks are outer tasks. In those scenarios DF2-LS is a bit slower than LBS, but its superior scalability justifies its use over LBS.

	BF-LS	**DF2-LS**	**LBS**	**AP**	**SP**
BF-Thefts	**Yes**	No	No	**Yes**	**Yes**
Lazy	**Yes**	**Yes**	**Yes**	No	No
Cost/Push	Low+ϵ	**Low**	**Low**	**Low**	**Low**
#Thefts	**Low**	Medium	Very High	**Low**	**Low**
#Pops	**Low**	Medium	**low**	High	Very High

Table 4.12: Comparison of schedulers.

Table 4.12 presents a high-level comparison of the four schedulers we will compare in the next section, including TBB's simple-partitioner, which is also used by Cilk++. We used a bold font to highlight the good qualities of each scheduler. The number of thefts and the number of pops (reclaiming work from one's own deque), are a measure of wasted overheads and should be minimized.

The table shows that BF-LS is the best approach, but it incurs slightly more overhead per push by accessing the task at the head of the list of postponed tasks, instead of pushing the current task like all the other compared schedulers do. Nevertheless, BF-LS is the only scheduler that minimizes both the number of thefts and the number of pop operations, so it is likely to be the best choice for performance and performance-portability. The experimental results in the next section support this hypothesis.

4.7 Experimental Evaluation of Lazy Work Stealing on Multicores

In this section, we start by showing that our proposed solutions, BF-LS and DF2-LS, we amend the scalability issues of LBS on the QUEENS benchmark presented in Section 4.5, and we count the number of thefts incurred by the different schedulers to show that our hypothesis that the depth-first lazy schedulers do indeed cause significantly more thefts was correct. Then, we evaluate BF-LS and DF2-LS on a set of benchmarks on three significantly different multicore platforms and show their performance improvement over LBS and TBB's default scheduler, auto-partitioner. Furthermore, we compare the software optimality of the compared schedulers on declarative code and on code that has been statically coarsened (manually or otherwise) to amortize scheduling overheads. Finally, we repeat the experiment of Section 3.4.2 and compute the worst-case software optimality of *QUEENS* using BF-LS. We show that BF-LS achieves better worst-case software optimality than AP with less manual coarsening.

4.7.1 Scaling of Lazy Scheduling on Multicores

First, we increment the results of Section 4.5 that illustrated the scaling problems of LBS, with the results for BF-LS and DF2-LS. To do that, we implemented those two alternative schedulers within TBB. Given LBS, the additional effort to implement DF2-LS, was trivial, and the effort for BF-LS was relatively modest. As before, we present the average of ten executions for each data-point. Speedups were computed relative to the execution time of an *optimized sequential version of the program*, as opposed to the execution of the parallel code on one worker. The standard deviations were shown in Tables 4.9, 4.10, and 4.11.

The improvement on T2 was very significant (Figure 4.10): BF-LS and DF2-LS achieved speedups of 10.8 and 10.5 compared to the speedups of 2.1 for LBS and 3.0 for AP. Another interesting trend was that DF2-LS performed better than

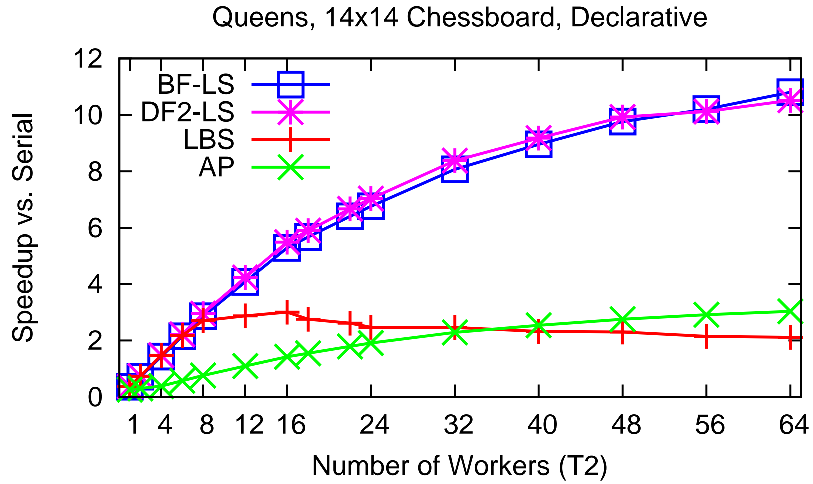


Figure 4.10: Scaling of schedulers on T2 (Queens)

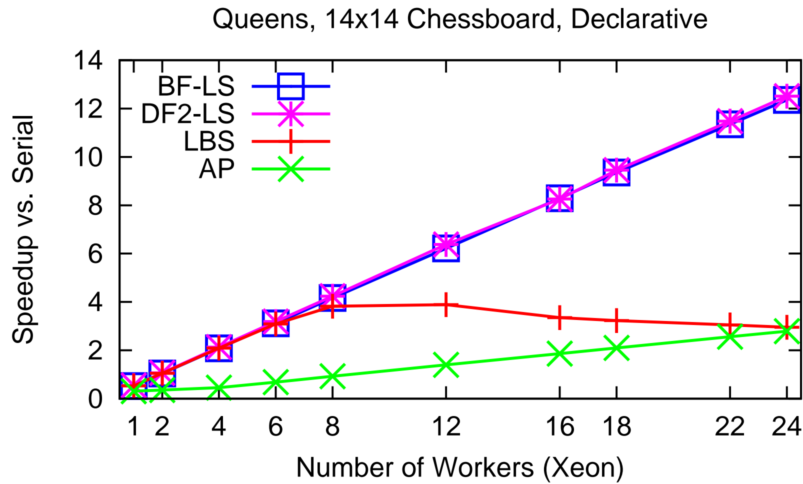


Figure 4.11: Scaling of schedulers on Xeon (Queens)

BF-LS on up to 56 workers, while BF-LS came ahead when using more workers. This illustrates two things: *(1)* BF-LS has a higher scheduling overhead per-task because it keeps track of postponed task descriptors, which causes it to fall slightly behind for smaller worker counts, and *(2)* for a large number of workers, DF2-LS starts suffering from the same scaling issues as LBS because it pushes the innermost postponed work. The same trends are observed on the Xeon (Figure4.11). BF-LS and DF2-LW achieve speedups of 12.5 and 12.4, whereas LBS and AP only reach

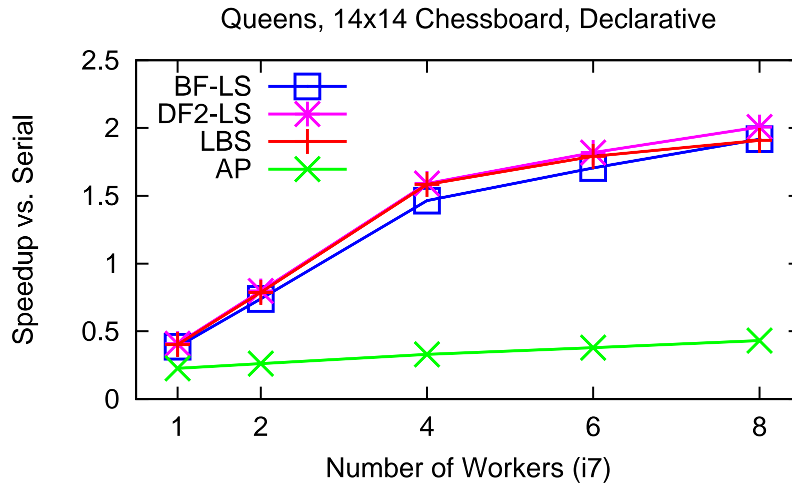


Figure 4.12: Scaling of schedulers on i7 (Queens)

3.0.

On the small platform, the i7 (Figure4.12), BF-LS and DF2-LS achieve a small performance boost relative to LBS only for the maximum number of threads (8), which supports the claim that, for small machines, LBS strategy of pushing the innermost tasks does not cause an excessive number of thefts, to the point of hurting performance.

On all three platforms, all three of TBB's available eager binary splitting schedulers (i.e., simple-partitioner, auto-partitioner, and affinity-paritioner), fall *significantly* behind BF-LS on the declarative version of QUEENS. Of the three TBB schedulers, only auto-partitioner is shown in the above figures as it consistently achieved better performance than simple-partitioner and affinity partitioner.

4.7.2 Counting Thefts

As we argued earlier, the limited scalability of LBS is caused by the choice of pushing the innermost postponed tasks, resulting in a larger number of thefts. We measured the number of thefts incurred by the four competing approaches on our three platforms. Table 4.13 displays the cumulative number of thefts performed

by all workers, averaged over 10 runs. The number of thefts with LBS is orders of magnitude larger than with all other approaches, and the number with DF2-LS si much larger than with BF-LS or AP. Finally, the number for BF-LS roughly matches that of AP. We believe that AP incurs slightly more thefts than BF-LS in this example because it runs much longer, as it wastes time pushing and popping tasks from the local deque. In an effort to load-balance over this longer execution time, some additional thefts occur.

Platform	**LBS**	**DF2-LS**	**BF-LS**	**AP**
T2(64)	55,593,973.2	1,045,072.0	3,130.5	3,303.2
Xeon(24)	4,161,559.1	10,562.9	791.9	906.6
i7(8)	316,337.6	973.8	228.1	274.1

Table 4.13: Number of thefts (Average over 10 runs)

4.7.3 Evaluation on a set of benchmarks

Queens was an enlightening toy example to experimentally demonstrate how LBS fails to scale up to a large number of workers, but it gives little confidence that using BF-LS as the default scheduler instead of AP or SP is a good idea. To address this, we compared the different approaches over the set of benchmarks summarized in Table 4.14. We selected these benchmarks because they exhibit a variety of computation and communication patterns [7]. This is important since we want to support general-purpose parallel code. Moreover, we needed benchmarks with nested parallelism to ensure scaling under composition and to expose the limitations of AP and SP.

Declarative	**Dataset (TBB)**	**Dataset (XMT)**	**Grain**	**Nesting**	**DOP**	**Work/Task**	**Description**
QUEENS(decl)	N = 14 Nodes	N = 11 Nodes	1	N	O($N!$)	O(1)	fine/irregular
TSP(decl)	N = 12 Nodes	N = 11 Nodes	1	N	O($N!$)	O(1)	fine/irregular
SpMV(decl)	80Kx5K, 40M non-zero	same	77	2	$40M$	O(1)	fine/irregular
BFS	10K Nodes, 8M Edges	same	53	2	O($\frac{\lvert E\rvert}{Diameter}$)	O(1)	fine/irregular
FW	N = 512 Nodes	same	91	1 (2D)	N^2	O(1)	fine/regular
Coarse(ned)	**Dataset(TBB)**	**Dataset(XMT)**	**Grain**	**Nesting**	**DOP**	**Work/Task**	**Description**
QUEENS(cut)	N = 14 Nodes	N = 11 Nodes	1	$^N/_2 = 7$	O($\frac{N!}{(N/2)!}$)	O($\frac{N}{2}!$)	coarse/irregular
TSP(cut)	N = 12 Nodes	N = 11 Nodes	1	$^N/_2 = 6$	O($\frac{N!}{(N/2)!}$)	O($\frac{N}{2}!$)	coarse/irregular
SpMV(coarse)	80Kx5K, 40M non-zero	same	1	1	$80K$	40M/80K=500	medium/irregular
MM	1024x1024 (N^2)	512x512	1	1 (2D)	N^2	O(N)	coarse/regular
CONV	4Kx4K (N^2) image, 16x16 (M^2) filter	1Kx1K, 16x16	1	1 (2D)	N^2	O(M^2)	coarse/regular

Table 4.14: Benchmark Summary

To quickly refresh the reader's memory, *TSP* is the traveling salesman problem on a dense graph, *QUEENS* finds all possible solutions to placing N queens on an N by N chessboard, *BFS* is breadth first search over a sparse graph, *SpMV* is sparse matrix by (dense) vector multiplication, *FW* is the Floyd-Warshall all-pairs shortest path on a dense graph, *MM* is the naive (N^3) dense matrix multiplication, and *CONV* is an image by filter convolution.

TSP(cut) and *QUEENS(cut)* are coarsened versions with manual parallelism cut-off after depth $N/2$. *SpMV(coarse)* computes each row of the sparse array sequentially, whereas *SpMV(decl)* uses TBB's parallel-reduce construct to expose all the parallelism (one task per non-zero element of the sparse array) and to efficiently aggregate the results. For that to be possible, we extended TBB's reduction operation to make it lazy and support the LBS, DF2-LS, and BF-LS schedulers.

The benchmarks are divided into *declarative*, where all the parallelism has been exposed, and *coarsened*, where either some of the parallelism was manually hidden (e.g., cut-off for *TSP* and *QUEENS*), or not all parallelism was exposed.

All the coarsened benchmarks can be re-written as declarative ones with $\mathrm{O}(1)$ work per task. For example, *MM* and *CONV* can be further parallelized using a parallel reduce operation, but due to their more regular nature, it is unlikely that the additional parallelism would reduce the run-time. Since this parallelization could be viewed as unnatural and unnecessary, we opted against it. However, in extreme cases where, for example, the multiplied arrays are vectors and the result is a single value, using the parallel reduce operation may be the only way to achieve a speedup.

For some benchmarks, we had to use a smaller dataset for the experiments on XMT (Section 4.4), because it has less memory than the multicores. The *grain* column shows the profitable parallelism threshold (ppt) taken from the XMTC compiler, the *nesting* column is the nesting depth of parallelism, and *DOP* is the degree of parallelism, i.e., the maximum number of tasks that can be executed in parallel, or, in other words, the maximum width of the computation DAG (directed acyclic

graph). For *BFS* it is on average in the order of the number of edges divided by the graph diameter. For our dataset, the diameter is 4. The next column represents the work per task, which happens to be constant for our declarative benchmarks and non-constant for our coarsened benchmarks.

TBB also provides *range* objects that describe 2D and 3D iteration spaces and which can be used to effectively flatten nested parallelism for dense, affine matrix computations. This allows exposing a multi-dimensional range of parallelism, whilst avoiding the use of nested parallelism, for which AP and SP are not very good. 2D and 3D range objects are not available in the XMT implementation (Section 4.8), so nested parallel loops are used on XMT.

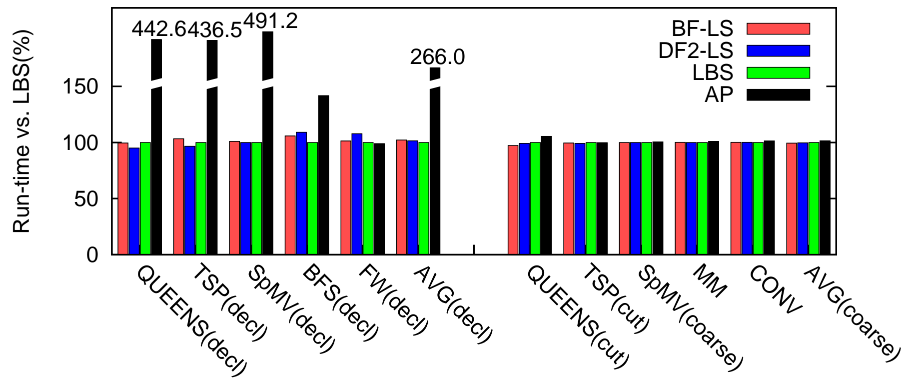


Figure 4.13: Benchmarks on the i7 using all 8 Workers

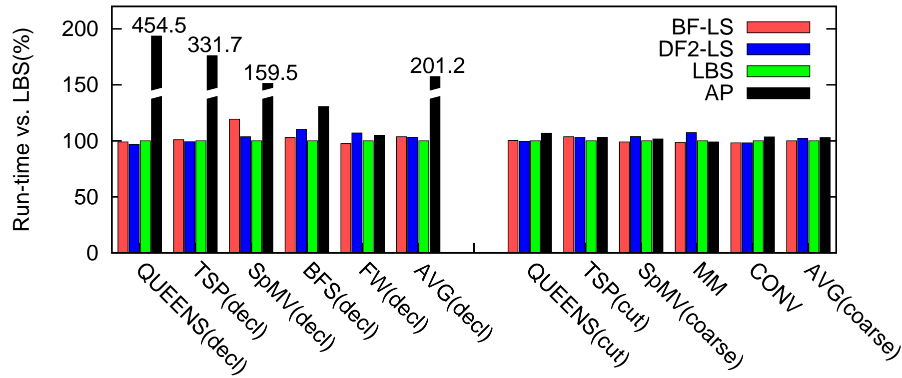


Figure 4.14: Benchmarks on the Xeon using only 6 Workers

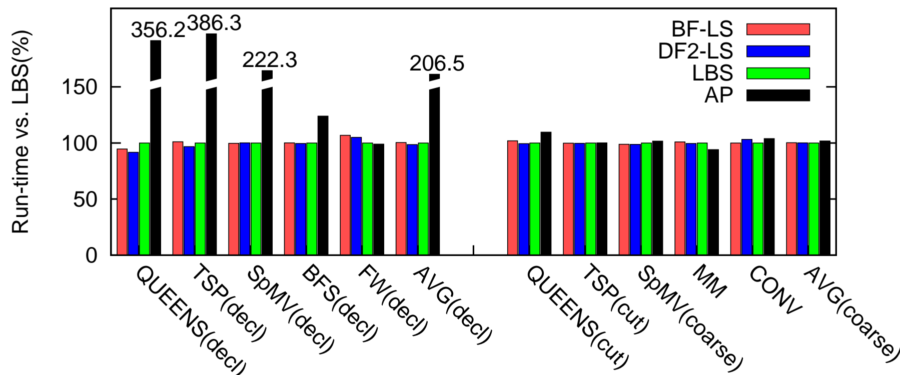


Figure 4.15: Benchmarks on the T2 using only 8 Workers

Declarative	BF-LS	DF2-LS	LBS	AP
QUEENS(decl)	0.49	1.19	2.44	0.67
TSP(decl)	0.91	1.02	1.19	0.27
SpMV(decl)	0.52	0.68	0.58	1.18
BFS(decl)	0.36	0.23	0.16	0.53
FW(decl)	0.17	0.13	0.37	0.08
Coarse	BF-LS	DF2-LS	LBS	AP
QUEENS(cut)	0.51	0.34	0.65	0.63
TSP(cut)	0.95	0.37	1.09	0.93
SpMV(coarse)	0.10	0.12	0.16	0.23
MM	0.07	0.04	0.08	1.38
CONV	0.33	0.64	0.62	1.71

Table 4.15: Standard Deviation(%) for i7 (Figure 4.13.)

Comparisons: Figures 4.13, 4.14, 4.15, 4.16, and 4.17 show the results on our three machines, grouped into declarative and coarsened benchmarks. We used the average of ten runs for the plots and the standard deviations are shown in Tables 4.15, 4.16, 4.17, 4.18, and 4.19. We used the geometric mean to compute the averages since we are averaging percentages (scaled values for the execution time). Our main goal was to demonstrate the superior performance of BF-LS and DF2-LS compared to LBS, but since LBS and AP have neither been compared on declarative code nor on multicores, the performance of AP was included in our performance figures for reference. Compared to AP (TBB's default scheduler), all three lazy approaches are faster on declarative code, and competitive on coarsened code. We also measured the performance of TBB's simple-partitioner and affinity-partitioner, but

Declarative	BF-LS	DF2-LS	LBS	AP
QUEENS(decl)	0.26	0.15	0.70	0.95
TSP(decl)	0.40	0.14	0.37	0.51
SpMV(decl)	14.48	10.94	6.36	11.78
BFS(decl)	7.82	10.04	11.28	6.46
FW(decl)	1.98	1.33	2.00	0.90
Coarse	BF-LS	DF2-LS	LBS	AP
QUEENS(cut)	0.14	0.14	0.14	0.09
TSP(cut)	0.13	0.08	0.13	0.12
SpMV(coarse)	7.05	11.02	12.91	15.15
MM	23.64	19.22	15.52	12.30
CONV	0.91	0.97	0.84	1.46

Table 4.16: Standard Deviation(%) for Xeon with 6 workers (Figure 4.14.)

Declarative	BF-LS	DF2-LS	LBS	AP
QUEENS(decl)	0.07	0.13	2.28	0.20
TSP(decl)	0.09	0.05	0.52	0.05
SpMV(decl)	0.11	0.15	0.15	0.37
BFS(decl)	0.18	0.17	0.16	0.15
FW(decl)	0.06	0.07	0.07	0.08
Coarse	BF-LS	DF2-LS	LBS	AP
QUEENS(cut)	0.20	0.15	0.29	0.20
TSP(cut)	0.05	0.04	0.09	0.04
SpMV(coarse)	0.12	0.15	0.16	0.69
MM	0.69	1.24	1.41	2.62
CONV	0.06	0.08	0.10	4.35

Table 4.17: Standard Deviation(%) for T2 with 8 workers (Figure 4.15.)

because auto-partitioner was consistently the best choice, we only compare against it.

On small size machines, such as the 4-core/8-thread i7 (Figure 4.13), the additional overheads of BF-LS and DF2-LS outweigh the benefits of incurring fewer thefts and LBS is only marginally faster. We also performed the same comparison on the Xeon machine using 6 workers (Figure 4.14) and on the 8-core T2 using 8 workers (Figure 4.15). These represent small multicore platforms. The conclusion is the same for all three small platforms: the three lazy approaches perform equally well, therefore LBS is preferable because it is the simplest to implement. The standard deviations for the 6-worker Xeon and the 8-worker T2 configurations

are shown in Tables 4.16 and 4.17. Notice that the 6-worker Xeon configuration has very high variation. We believe the reason for this high variability is that workers are not pinned to cores and the operating system (OS) naively migrates them across chips, causing them to lose their cached values. Whatever the reason, the results in Figure 4.14 are unreliable, but the ones in Figures 4.13 and 4.15 have low variability and are reliable.

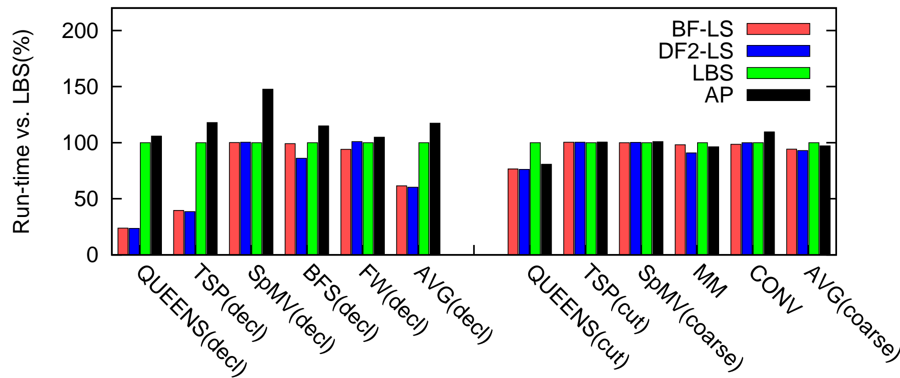


Figure 4.16: Benchmarks on the Xeon using all 24 Workers

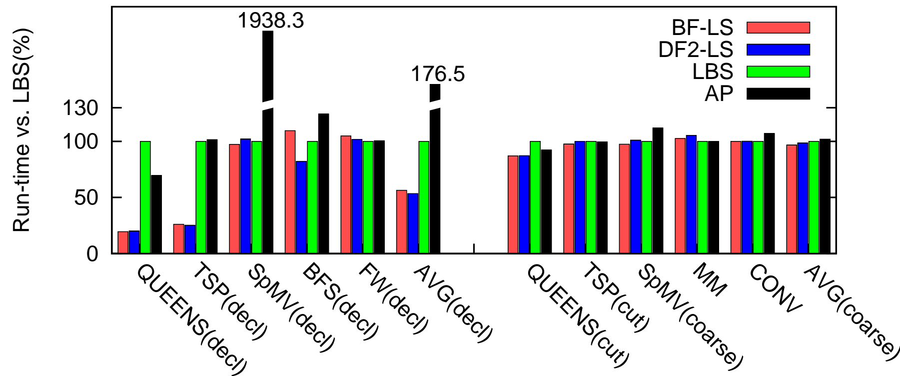


Figure 4.17: Benchmarks on the T2 using all 64 Workers

On the larger machines the situation is much different. On the 24-core Xeon (Figure 4.16), LBS fails to scale on the recursively nested declarative benchmarks *TSP(decl)* and *QUEENS(decl)* and falls behind on *BFS* and *QUEENS(cut)*. DF2-LS is the best scheduler, outperforming on average the other three approaches both on declarative (by 47.1% vs. LBS) and on coarsened benchmarks (by 5.7% vs. LBS). BF-LS is a close second and outperforms LBS by 44.6% on declarative benchmarks

Declarative	BF-LS	DF2-LS	LBS	AP
QUEENS(decl)	1.48	0.43	3.39	0.62
TSP(decl)	1.05	0.65	3.17	0.50
SpMV(decl)	0.76	1.04	0.37	2.80
BFS(decl)	0.66	0.62	0.62	0.39
FW(decl)	1.98	2.17	1.94	1.47
Coarse	BF-LS	DF2-LS	LBS	AP
QUEENS(cut)	1.16	0.74	0.78	0.54
TSP(cut)	1.15	1.04	1.09	0.98
SpMV(coarse)	0.37	0.27	0.30	0.33
MM	6.10	0.62	4.17	2.14
CONV	1.01	0.81	0.81	2.02

Table 4.18: Standard Deviation(%) for Xeon with 24 workers (Figure 4.16.)

Declarative	BF-LS	DF2-LS	LBS	AP
QUEENS(decl)	0.27	2.00	6.82	1.17
TSP(decl)	0.30	0.47	2.88	0.10
SpMV(decl)	0.09	0.20	0.12	0.12
BFS(decl)	0.21	0.49	0.26	0.29
FW(decl)	0.12	0.13	0.16	0.11
Coarse	BF-LS	DF2-LS	LBS	AP
QUEENS(cut)	0.08	0.48	1.05	0.05
TSP(cut)	0.46	0.25	0.98	0.70
SpMV(coarse)	0.23	0.24	0.10	0.87
MM	2.56	2.06	3.93	1.43
CONV	0.06	0.08	0.08	2.17

Table 4.19: Standard Deviation(%) for T2 with 64 workers (Figure 4.17.)

and by 5.8% on the coarsened ones. On the 64-thread T2 (Figure 4.17), DF2-LS and BF-LS are comparable, with DF2-LS being a lightly better on declarative codes, but slightly worse on coarsened codes. BF-LS is 51.7% faster than LBS on declarative code, and 1.9% on coarsened.

BF-LS falls behind DF2-LS on BFS on both lager-sized platforms. We believe this to be due to the higher scheduling overhead per-task of BF-LS compared to DF2-LS or LBS, thereby requiring a higher *ppt* coarsening threshold. However, we did not grant BF-LS a higher *ppt* to keep comparisons simpler.

Scaling up: We expect that on larger platforms the performance of DF2-LS will suffer just like LBS, because it also violates the breadth-first theft principle of work stealing. Figure 4.18 shows the performance scaling of the three lazy schedulers on the declarative version of computing the 36th Fibonnaci number. This computation is notorious for its high communication to computation ratio, which stresses the capabilities of schedulers. Once again, we used the average of the runs, and the standard deviations are shown in Table 4.20.

As expected, LBS does not scale well. Similarly, DF2-LS gradually stops scaling for larger numbers of workers. With more than 32 workers, BF-LS becomes the best approach because it scales better, but for fewer workers, its higher per-task overhead makes it fall behind.

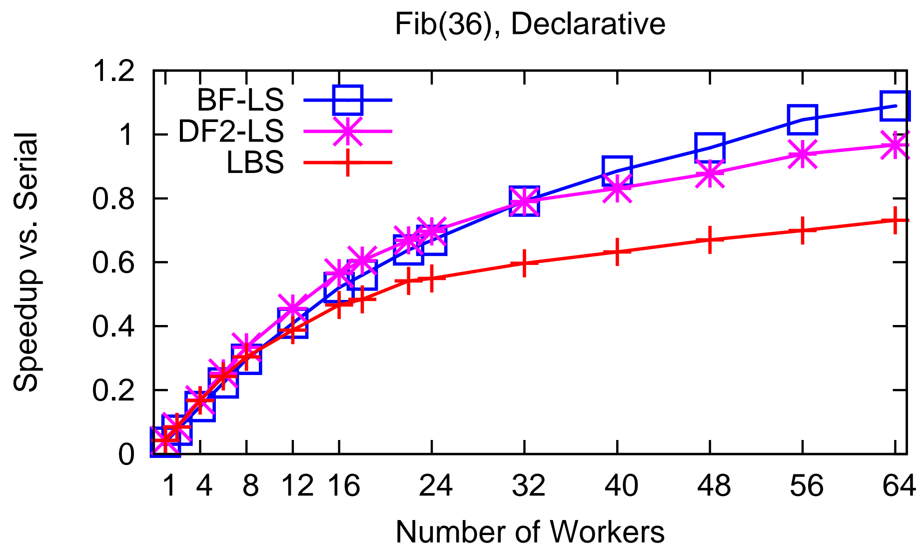


Figure 4.18: Scaling of schedulers on T2 (Fib(36))

The low speedup numbers in Figure 4.18 are not surprising since we did not implement any manual cut-off and because TBB's overheads for creating a task descriptor are relatively high. This high overhead is mainly due to TBB being implemented as a library and having to create several objects to call the scheduler. However, the goal here is to show how the performance of the different schedulers

# **Workers**	BF-LS	DF2-LS	LBS
1	0.26	0.12	0.10
2	0.26	0.17	0.19
4	0.13	0.07	0.27
6	0.07	0.15	0.76
8	0.18	0.51	1.10
12	0.13	0.88	1.77
16	0.21	0.89	1.20
18	0.16	0.82	1.59
22	0.29	1.66	2.26
24	0.70	1.67	2.12
32	1.54	1.39	2.23
40	0.30	2.37	3.43
48	1.19	1.12	3.09
56	1.04	0.88	3.29
64	1.01	3.03	2.62

Table 4.20: Standard Deviation(%) for T2 (Figure 4.18.)

scales under extreme stress, to try and emulate the stress of running on a larger machine. To that effect, we think that these results provide some insight.

Given the above numbers, our recommendation would be to always use BF-LS because its performance is more robust than the depth-first lazy approaches (DF2-LS and LBS) and because it does not fall significantly behind the best approach in the few cases when it is not itself the best scheduler. This gives us greater confidence that, if used on other benchmarks and applications than the ones presented here, BF-LS will not surprise with lower than expected performance. Moreover, if one wants to create a binary of their parallel application to be executed on platforms of varying sizes, BF-LS is the best choice.

4.7.4 Software Optimality of Declarative Code

One of the claims we have made is that our work on lazy scheduling brings us one step closer to efficient execution of declarative code. We substantiate this claim by comparing the performance of declarative code to that of its coarsened counterpart. We use the definition of *software optimality* from Section 3.4 (Equation 3.1) and compare the software optimality of the different schedulers on declarative code.

Ideally, we should take the minimum over all possible coarsenings to properly define 100% software optimality, but given that the coarsenings we used were carefully selected to maximize performance, and that values of software optimality we present are all below 100%, the effort of trying all possible coarsenings to get a slightly more accurate lower-bound on execution-time was not justified.

i7	BF-LS	DF2-LS	LBS	AP
QUEENS	43.3	45.4	43.1	9.7
TSP	38.0	40.6	39.2	9.0
SpMV	95.2	96.0	96.0	19.6
AVG	53.9	56.1	54.6	12.0

Table 4.21: Software Optimality (%) of Declarative Code on i7

Xeon	BF-LS	DF2-LS	LBS	AP
QUEENS	56.5	57.2	13.5	12.7
TSP	45.5	46.9	18.0	15.3
SpMV	99.4	99.2	99.6	67.4
AVG	63.5	64.3	28.9	23.6

Table 4.22: Software Optimality (%) of Declarative Code on Xeon

T2	BF-LS	DF2-LS	LBS	AP
QUEENS	38.5	37.5	7.5	10.8
TSP	25.2	26.0	6.6	6.5
SpMV	86.2	82.2	84.0	4.3
AVG	43.7	43.1	16.1	6.7

Table 4.23: Software Optimality (%) of Declarative Code on T2

In Tables 4.21, 4.22, and 4.23, we present, for each of our three platforms, the software optimalities of the four compared schedulers for the three benchmarks for which we had both a declarative and a coarsened version. We used the geometric

mean to compute the average. The results show that LBS greatly improves the software optimality compared to AP on small platforms (i7), but fails to deliver for the deeply nested benchmarks on larger platforms (*TSP* and *QUEENS*. On the other hand, BF-LS and DF2-LS dramatically improve the software optimality compared to AP for all benchmarks on all platforms, achieving over 50% software optimality on average on i7 and Xeon, and over 6 times the software optimality of AP on T2.

When the compiler is able to perform coarsening to amortize the overheads per task, such as for *SpMV*, the software optimality of our proposed solutions becomes competitive with that of manually coarsened code, but without compromising the performance portability. This is an indication that, with BF-LS, the programmer will no longer need to prune the exposed parallelism, which is complicated and hurts performance portability.

On the other hand, when the compiler is unable to perform coarsening to amortize the overheads per task, such as for *TSP* and *QUEENS*, the software optimality is low. Even so, novice programmers will not be discouraged, as their first parallel implementations will achieve significant speedups.

4.7.5 Software Optimality of Code with Amortizing Coarsening

In this section, we ask how much software optimality can be achieved using the lazy schedulers, assuming that the compiler or the programmer have performed coarsening to amortize scheduling overheads. To do so, we repeat the previous experiment, but this time, we add a cut-off depth for *QUEENS* and *TSP* and use the profitable-parallelism threshold that was automatically computed for the lazy schedulers with AP as well. The cut-off depth is such that, if the sub-problem would not benefit from parallelization, it is solved sequentially. In other words, we find the profitable parallelism threshold as a cut-off depth.

To find the cut-off depth for *QUEENS*, we run increasing input sizes $n \in \{1, 2, 3, \ldots\}$ and measure the sequential execution and the parallel execution on two

workers with the cut-off depth equal to 1. Let k be the the minimum n for which $ParTime(n) < SerTime(n)$; the cut-off function will be $N - (k-1)$, in other words $k - 1$ recursive levels from the bottom of the recursion. We repeat this process on each target platform and get a different value of k for each of them. We repeat the same procedure for *TSP* and find the amortizing cut-off depths for each of our multicore platforms.

Table 4.24 shows the values of $k - 1$ for each of the platforms and benchmarks. There is little variation of the profitable parallelism cut-off depth between platforms, as only the cut-off for *TSP* on the i7 is lower. This is because, due to its smaller scale, the i7 can profit from smaller granularity of parallelism.

	T2	Xeon	i7
QUEENS	5	5	5
TSP	5	5	4

Table 4.24: Amortizing Cut-Off Depths for QUEENS and TSP

Tables 4.25, 4.26, and 4.27 show the software optimality results of the different schedulers with amortized code. BF-LS is the clear winner in this comparison, with average software optimality results between 90.4% and 95.6%. Auto-partitioner is the worst with average software optimality results between 60.8% and 74.9%. LBS also falls significantly behind BF-LS on the two larger platforms, Xeon and T2.

These results show two things. First, our proposed separation of coarsening responsibilities of amortizing scheduling overheads and pruning parallelism is not artificial, since AP was not able to keep up with the lazy schedulers when parallelism was statically coarsened to amortize scheduling costs. In other words, pruning parallelism is beneficial even when coarsening to amortize scheduling overheads has been performed. Second, lazy scheduling constitutes a significant step towards supporting declarative code because it achieves very high software optimality on irregular codes that have been coarsened just enough to amortize scheduling overheads.

i7	BF-LS	DF2-LS	LBS	AP
QUEENS	88.6	90.3	86.4	46.7
TSP	93.4	94.6	93.1	89.2
SpMV	95.2	96.0	96.0	53.9
AVG	92.3	93.6	91.8	60.8

Table 4.25: Software Optimality (%) of Amortized Code on i7

Xeon	BF-LS	DF2-LS	LBS	AP
QUEENS	88.1	83.7	34.1	52.5
TSP	99.8	98.4	97.4	98.9
SpMV	99.4	99.2	99.6	81.1
AVG	95.6	93.5	69.1	74.9

Table 4.26: Software Optimality (%) of Amortized Code on Xeon

T2	BF-LS	DF2-LS	LBS	AP
QUEENS	85.6	84.6	47.6	53.3
TSP	100.0	96.0	97.1	93.3
SpMV	86.2	82.2	84.0	54.9
AVG	90.4	87.4	72.9	64.9

Table 4.27: Software Optimality (%) of Amortized Code on T2

4.7.6 Worst-Case Software Optimality of Lazy Scheduling

In this section, we revisit the experiment of Section 3.4.2, but this time, we wanted to measure the worst-case software optimality of breadth-first lazy work stealing (BF-LS) on *QUEENS*. We evaluate three cut-off functions: $N-5$, $N-6$, and no cut-off. The results are shown in Table 4.28.

With a cut-off depth of $N-5$, we achieved better worst-case software optimality with BF-LS than AP achieved with the more complex and aggressive coarsening function of $\min(N/3, N-5)$. The combination of the two functions, $N/3$ and $N-5$,

$SwOpt_{WC(W_{T2})}(QUEENS, c, BFLS; i)$

	Size of N (Side of the Board)						
Depth	4	6	8	10	12	13	$SwOpt_{WC}(c)$
$N-5$	100.00	86.82	83.60	84.68	78.95	77.53	77.53
$N-6$	100.00	57.51	47.45	97.06	92.66	91.27	47.45
NoCut	19.70	12.03	17.42	20.66	24.52	26.93	12.03

$\max_{c \in \mathbb{C}} SwOpt_{WC}(QUEENS, c, LBS) = 77.53\%$, with $c = N - 5$.

Table 4.28: Worst-Case Software Optimality for BF-LS.

attempted to cover both goals of coarsening – amortizing scheduling overheads and pruning parallelism. Since BF-LS takes care of pruning parallelism, it only needs a cut-off function of $N-5$, which was selected because only for sizes of N larger than 5 does executing the computation in parallel become beneficial on the T2 platform (see Table 4.24).

Moreover, the worst-case software optimality of BF-LS without any manual coarsening is significantly better than that of AP (12% vs. 3.5%, cf. last line of Table 3.3). All these results suggest that, for constant programmer coarsening effort, BF-LS delivers better performance than AP and SP, the state-of-the-art schedulers used today to schedule task-parallel codes.

These results suggest that, with BF-LS, the programmer needs to spend less time fine-tuning the granularity of tasks in their code, bringing us one step closer to efficient support of declarative code. Furthermore, using BF-LS gives the code superior performance portability than AP (and SP), as shown by the superior worst-case software optimality over a set of inputs and platform subsets.

Of course, these results are not conclusive, since only one benchmark was evaluated on one platform. To obtain more results, a tool for automating the execution of these experiments and collecting the results is would be valuable, since they involve so many different combinations of coarsenings, inputs, platforms, and

platform subsets. To run these experiments for small inputs, we manually determined how many times the computation needs to be repeated in a loop, in order to keep the execution times short, while ensuring that the aggregate running time remained significant enough to get precise timings, by drowning the noise from external events (e.g., OS context switches). We also took into account the subset of the platform used: when using a single worker we needed fewer repeats than when using numerous workers. This tuning phase of the experiment would have to be automated by the envisioned tool to make it practical to get results with multiple inputs and platforms. However, building such a tool was beyond the scope of this dissertation.

4.8 Experimental Evaluation of Scalable Lazy Work Stealing on XMT

In this section, we repeat the comparison of the four schedulers (BF-LS, DF2-LS, LBS, and AP) on the XMT manycore architecture prototype developed at the University of Maryland. We do that *(1)* because one of our goals is to be able to compile and execute declarative code efficiently on various general-purpose platforms, *(2)* because XMT seems to allow LBS to scale well, and *(3)* because XMT supports efficient execution of fine-grained irregular parallelism, which is needed for declarative codes.

Earlier (and in [83]), we evaluated LBS on XMT, but focused on speedups rather than on efficiently supporting declarative code. So for *QUEENS* and *TSP*, the parallelism was manually cut-off after half of the recursive depth. This optimization suffices to hide the scaling issues of LBS, even on traditional multicores, as shown in Figures 4.16 and 4.17, so we ran the declarative version of *QUEENS* on XMT and were surprised to find that LBS scaled well (Figure 4.19).

4.8.1 Scaling of Lazy Scheduling on XMT

Two hardware peculiarities of XMT are likely to help LBS scale well on XMT. First, *hardware-assisted scheduling* efficiently distributes in parallel tasks coming from the outermost collection of parallel tasks (e.g., parallel loop) to workers requesting work. LBS and the other software schedulers are only used to schedule nested parallelism. The effect of combining the hardware and software scheduling is that the hardware makes all of the outer parallelism available, and LBS deploys a large portion of the tasks of the first nested level before getting deeper in the recursion. This increased availability of shallow parallelism on the deques mitigates the harmful effect of pushing deeply nested tasks later on, much like DF2-LS. Second, XMT does not have private caches to avoid coherence issues. Consequently, thefts do not generate coherence traffic, like they do on multicores where a cache-line holding part of the victim deque has to be modified by the thief. Hence, thefts cost approximately as little as popping from the local deque, decreasing the sensitivity of performance to their number.

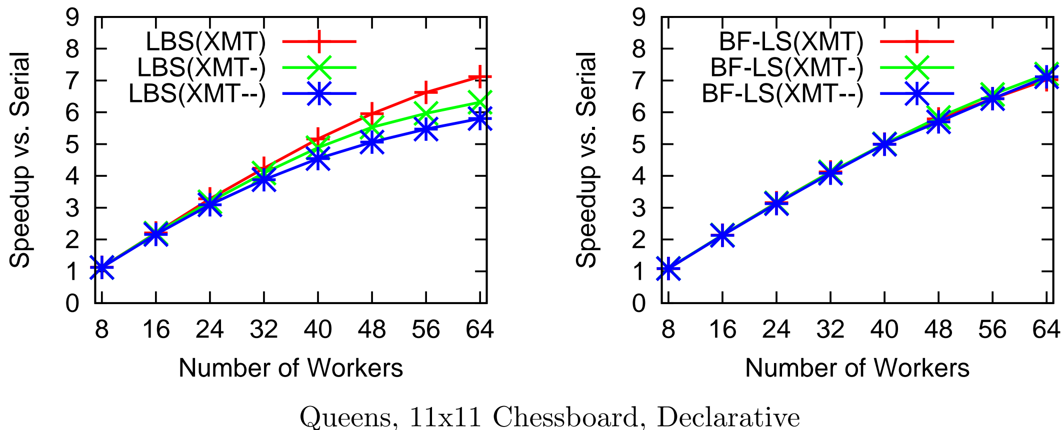


Figure 4.19: Scalability of LBS & BF-LS on XMT, XMT- and XMT--.

To determine if the hardware scheduler is helping LBS scale to 64 TCUs, we disabled it by adding to the code an artificial outer parallel loop of a single iteration that does nothing more than run the actual parallel code. This forces the hardware

to schedule the dummy iteration, and lets LBS (and other software methods) schedule the entire parallel computation. *We call this configuration without the hardware scheduler XMT-.* In Figure 4.19, it is evident that LBS does not scale as well on XMT- as it does on XMT, confirming our hypothesis that the hardware scheduler helps LBS. Nevertheless, we do not see the same drastic loss of performance on XMT as on multicores (Figure 4.10), suggesting that XMT's hardware scheduler is not the only mechanism helping LBS to scale.

To test whether the lack of coherence on XMT helps LBS scale, we simulated coherence traffic induced by thefts by adding an artificial delay for thefts between discovering a non-empty victim deque and attempting to steal from it. The artificial delay is a sequential busy loop of 1,000 iterations, which is intended to also simulate the data and code transfer for the stolen task. *We call this configuration with the simulated coherence and the disabled hardware scheduler XMT--.* Figure 4.19 demonstrates that LBS is sensitive to the cost of thefts whereas BF-LS is not, which aligns with our expectation that LBS incurs more thefts. Nevertheless, we have yet to observe the drastic loss of performance displayed by LBS on multicores. It is an interesting question for future work to determine the precise reasons for this apparent superior scaling of LBS on XMT and XMT--, as the answer could guide the design of future multicores.

He et al. [45] assume that a theft costs 15,000 instructions, which is much higher than our artificial delay of 1,000 iterations that translate to about 2,000 instructions with register operands. It would be interesting to increase the artificial delay to 15,000 instructions in order to evaluate the scalability of LBS on XMT with that higher delay. Note, however, that this cost of 15,000 instructions for thefts was chosen as a conservative upper-bound by He et al., and not as an accurate estimate of the actual cost of thefts.

4.8.2 Counting Thefts

We also tried to measure the number of thefts on XMT, but were unable to add code to count the thefts without greatly affecting the execution time, and thus the number of thefts. At the same time, using the XMT simulator [54] to count the number of thefts for computations of this scale was not possible, due to the long simulation time. For that reason, we do not know the number of thefts incurred by the different schedulers on XMT.

4.8.3 Evaluation on a set of Benchmarks

We also evaluated the four schedulers on our set of benchmarks on XMT and XMT- (Figures 4.20 and 4.21). We used the average of ten runs and the standard deviations are shown in Tables 4.29 and 4.30. Here again, we see that the hardware scheduler helps LBS, and that our proposed approach, BF-LS, scales robustly and is more platform-independent.

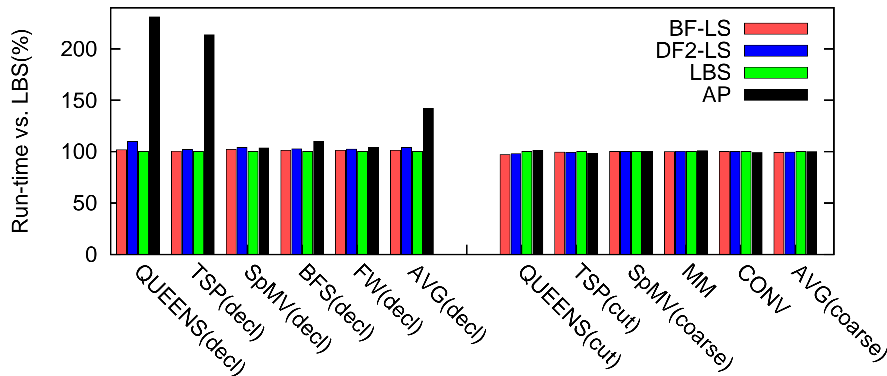


Figure 4.20: Benchmarks on XMT using all 64 TCUs.

On XMT (Figure 4.20), LBS is the best scheduler for the declarative benchmarks, closely followed by BF-LS and DF2-LS, while AP falls behind by 53.9%. On coarsened benchmarks, all approaches are equivalent on average.

On XMT- (Figure 4.21), on declarative benchmarks, DF2-LS is 8.2% faster

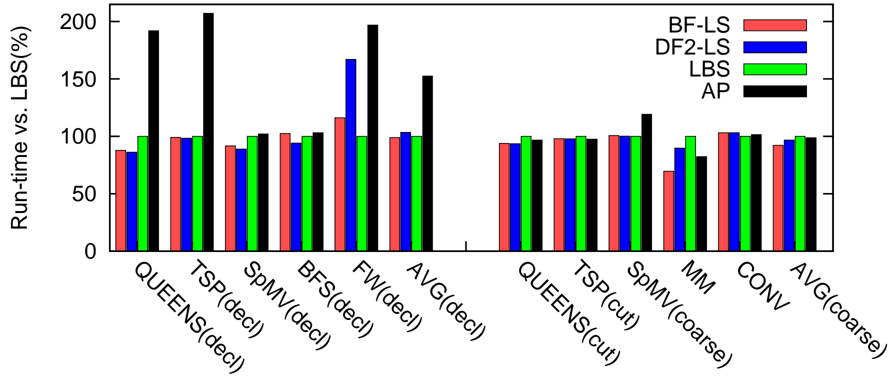


Figure 4.21: Benchmarks on XMT- using all 64 TCUs.

than LBS, and BF-LS is 4.9% faster than LBS. AP is 42.1% slower than LBS. On the coarsened benchmarks, BF-LS is 4.2% faster than LBS. DF2-LS and AP are slower than LBS by 6% and 10.8%. Note that the standard deviation of the measurements for SpMV(coarse) is high with AP because on XMT we are using a value of K=1 for AP, splitting the parallelism evenly into P tasks and then potentially doing more splitting. This results in load-imbalanced chunks and in high variability for SpMV(coarse) with AP, but on average, AP_{xmt} was preferable to $\text{AP}_{default}$ even on XMT- (i.e., with the hardware scheduler disabled). Therefore, we presented the results for AP_{xmt} where $K = 1$ and $V = 4$.

Here too, we see that, overall, BF-LS seems to be the best choice because it consistently performs well. We also observe that the hardware assisted scheduling benefits LBS by exposing more of the outer tasks for parallel execution. However, it is quite probable that, for larger instances of XMT, LBS will suffer from the same scaling issues we witnessed on multicores, despite the assistance from the hardware scheduler.

4.9 Related Work

In this section, we present previous work on two types of schedulers: *(1)* those that support parallel function calls or futures but not parallel loops, and *(2)*

Declarative	BF-LS	DF2-LS	LBS	AP
QUEENS(decl)	0.93	1.76	2.52	0.57
TSP(decl)	0.21	0.74	0.46	0.43
SpMV(decl)	0.01	0.01	0.01	0.01
BFS(decl)	0.11	0.05	0.05	0.11
FW(decl)	0.01	0.01	0.01	0.01
Coarse	BF-LS	DF2-LS	LBS	AP
QUEENS(cut)	0.57	0.67	1.71	0.30
TSP(cut)	0.35	0.20	0.34	0.03
SpMV(coarse)	0.01	0.01	0.01	0.01
MM	0.01	0.01	0.01	0.01
CONV	0.01	0.00	0.00	0.02

Table 4.29: Standard Deviation(%) for XMT with 64 workers (Figure 4.20.)

Declarative	BF-LS	DF2-LS	LBS	AP
QUEENS(decl)	1.50	1.55	1.92	0.80
TSP(decl)	0.42	0.52	0.48	0.26
SpMV(decl)	2.39	1.34	1.99	2.97
BFS(decl)	0.39	0.27	0.93	0.55
FW(decl)	0.56	0.39	0.11	0.56
Coarse	BF-LS	DF2-LS	LBS	AP
QUEENS(cut)	0.85	0.53	1.83	0.30
TSP(cut)	0.43	0.21	0.82	0.05
SpMV(coarse)	0.10	0.14	0.17	11.33
MM	3.52	3.22	1.91	5.73
CONV	0.47	0.28	0.22	0.24

Table 4.30: Standard Deviation(%) for XMT- with 64 workers (Figure 4.21.)

those that explicitly support parallel loops. Then, we present work on *throttling parallelism*: serializing parallelism at run-time to minimize overheads.

4.9.1 Schedulers without Parallel Loop Support

These approaches do not explicitly support parallel loops; instead they introduce parallelism through function calls or futures, one task at a time. Handling of parallel loops explicitly opens optimization opportunities not available to parallel function calls, since loops create many tasks simultaneously, instead of one at a time. Multiple tasks can be packaged into a single task descriptor, greatly reducing the number of deque transactions and leading to much better performance. Work

stealers that do not explicitly support parallel loops miss these optimization opportunities and deliver inferior performance. Eager Binary Splitting (AP and SP) is thus our primary competitor because it explicitly supports parallel loops. Nevertheless, methods for parallel function calls are outlined below, because they were the first results on work stealing and made it popular.

Work stealing has become popular in part because of its efficient implementation in the Cilk programming language [38]. The Cilk compiler creates two clones of functions, a fast and a slow clone. The fast one simply skips the synchronization of tasks with their continuation if they executed on the same worker (i.e., the continuation is not stolen). The slow clone is executed when the task is stolen and may have, therefore, executed concurrently with one of its siblings or children. This optimization is orthogonal to our proposed lazy scheduling, and the two should be combined for optimal performance. Cilk [38] was designed for parallel function calls (i.e. relatively coarse-grained parallelism), however, and it is not optimized for parallel loops. We have adopted a Cilk-like implementation that follows its work-first principle and called this implementation serializing work stealing (SWS). Our results showed that it performs much worse than Lazy Scheduling for parallel loops. This result is not surprising since Cilk was not meant for parallel loops. Other approaches that focus on coarser parallelism, such as parallel function calls and futures [56, 67, 41, 81], have the same limitations.

Arora et al.[6] propose a non-blocking implementation of work stealing which is well suited for multiprogrammed systems. Their approach suffers from deque overflows that can cause the program to crash. Two other approaches [26, 46] propose complicated solutions to the overflow problem. Lazy Scheduling sidesteps the problem of overflowing the deques since it will stop pushing task-descriptors on a deque that exceeds a threshold size. Therefore, deques are implemented as constant-size circular arrays and overflow is not an issue.

Acar et al. [2] describe a method to improve the locality of work stealing.

This approach is implemented in TBB [76] and called *affinity-partitioner*(AfP). We also compared our solutions to AfP and found it to be slower than auto-partitioner on average, which is why we excluded it from the presentation. In fact, we also implemented a lazy version of auto-partitioner, but it was also generally slower than our proposed solutions. We believe lazy auto-partitioner was slower because Lazy Scheduling relies on checking the deque frequently to push work for hungry workers, but auto-partitioner coarsens tasks into large chunks which prevents frequent checks.

Hendler et al. [47] propose to steal half the task descriptors off a deque instead of just one, so as to better spread the work across the system and prove good theoretical bounds for load-balance. Their approach is not applicable to Lazy Scheduling, however, because unless a higher threshold is selected for the deque size, each deque will have at most one (or two in the case of DF2-LS) task-descriptors at all times. In our experience, picking a higher threshold is detrimental to performance. However, in the case of parallel loops where binary splitting (lazy or eager) starts by pushing a task descriptor with half the tasks onto the deque, one could say that Hendler's advice to steal half of the remaining tasks is heeded. The lazy aspect of scheduling is an added benefit in addition to the binary splitting.

Goldstein et al. [41] propose a lightweight task creation mechanism for nested parallelism (they use the term *thread* for tasks). Their proposed approach has the same serializing problem as help-first work stealing because it relies on the parent task activating *nascent threads* upon request by a remote processor to make them available for execution on the remote processor.

4.9.2 Schedulers with Parallel Loop Support

When we began this work, the only work stealing schedulers that explicitly supported parallel loops were TBB's SP and AP [76], which is why they were the focus of our comparisons. As discussed, the programmer is expected to determine a good value for the stop-splitting-threshold of each parallel loop when using SP,

by trying out various values. Moreover, this fixed threshold limits the performance portability of the code to a different number of cores, datasets and contexts. Lazy Scheduling frees the programmer from choosing a threshold manually and adapts to run-time conditions to avoid excessive splitting without falling behind on performance. AP does not require programmer tuning, but it still falls behind Lazy Scheduling because it lacks context portability, as it does not perform run-time adaptive coarsening. Instead, it adaptively reverses some of the coarsening it does by default, which can already be insufficient in cases of nested parallelism.

Cilk++ [62] implements eager binary splitting using the simple-partitioner approach with a default stop-splitting threshold (i.e., grain-size) of 1 (SP_1), in the absence of a programmer supplied one. SP_1 falls significantly behind Lazy Scheduling on code with fine-grained parallelism. CilkPlus is the latest reincarnation of the Cilk language and follows the same approach as Cilk++ for parallel loops.

Guo et al. [43] present a scheduler that adaptively chooses between two work stealing approaches: work-first (which we call SWS) and help-first. In work-first, the worker picks the child task and places its continuation on the deque, whereas in help-first, it places the child task on the deque and executes the continuation. In the absence of parallel loops, choosing between the two approaches is orthogonal to Lazy Scheduling. If parallel loops are introduced, both approaches serialize parallelism creation and fork-off *grain* tasks at a time. In contrast, binary splitting approaches (eager or lazy) overcome this serialization and create task descriptors with more tasks, improving the load-balancing effect of thefts, as shown by Hendler [47].

Bergstrom et al. [13] combined Lazy Scheduling with *zippers*, an approach for splitting trees, which is how arrays are represented in their functional programming language. They call their approach Lazy Tree Splitting and show improved performance robustness across their benchmark suite compared to eager binary splitting with simple-partitioner.

The remaining schedulers in this sub-section support parallel loops but not

work stealing. OpenMP [72] recognizes the need for nested parallelism by providing primitives, but whether nesting is truly supported or not is implementation specific. Frequently, OpenMP implementations serialize inner parallelism, which our results show has serious performance limitations.

The nano-threads library supports nested parallelism [65] and can be used for OpenMP but uses a ready queue, or a hierarchical ready queue [71] for scheduling, both of which can have an arbitrarily higher memory footprint than work stealing. Additionally, access to the head or tail of a queue must be synchronized among all threads, and a hierarchical ready queue (a tree of queues) has a single enqueue point, the root, and requires multiple operations to get work to the leaves, where it is dequeued. This makes them unsuitable for our goal of supporting declarative fine-grained parallelism.

Duran et al.[35] propose a system that assigns processors to tasks by instrumenting the code and getting run-time statistics to refine the distribution. They assume, however, that the programmer has coarsened the outer parallelism into *ngroups* (similar to setting the *sst*), and has also defined the grain-size (*sst*) of the inner parallelism. Lazy Scheduling does not need to collect run-time statistics, and does not place the burden of coarsening on the programmer.

NESL[15] employs complex compiler transformations to support nested parallelism by flattening[14]. NESL is an interpreted functional language without side-effects, which limits its scope. Moreover, it is unclear if good performance can be achieved since only three benchmarks were evaluated (only one with nested parallelism) on three architectures, and in most cases, their approach falls behind native code for these machines. The claim is that performance would be significantly improved if the language were compiled instead of interpreted, but we are unaware of a study quantifying this claim. The approach of flattening nested parallelism seems less fit for multithreaded platforms, such as the ones work stealing targets, because it effectively tries to make some of the run-time scheduling choices at compile-time

with the limited information it has available, so as to partition the computation as evenly as possible to the processing units. Flattened code is, however, particularly important for the vector machines that were the basis of most supercomuters through the 80's and into the 90's when this work was published.

4.9.3 Parallelism Throttling

Kranz et al. [56] and Certner et al. [25] have also used run-time conditions to decide between creating more parallelism or executing work serially, but they rely on maintaining extra state (e.g., a global counter), which creates a hot-spot and does not scale well. Moreover, these approaches make irrevocable serialization decisions that may hurt load-balancing. Lazy Scheduling only postpones exposing parallelism to other workers and runs one or a few tasks before checking the system-load again.

Duran et al.[34] propose an interesting way to limit the creation of excessive parallelism, which is not related to scheduling. In fact, they experiment with several schedulers, demonstrating that their method works well with all of them. They inject code that collects statistics about the amount of work of different procedures as a function of the depth (of the call-stack) at which they are called. When enough statistics have been collected, they turn off this profiling, and use the information to decide which procedures to serialize and at what depth. Given a recursive parallel procedure such as quicksort, their approach will decide at which depth of the recursion to start calling a serial version of quicksort. This approach is orthogonal to Lazy Scheduling because it does not solve the need to schedule the work, and it can be applied on top of it. In fact, our coarsened recursively nested benchmarks (*TSP*, *QUEENS*, *QSort*) have manual parallelism cut-offs that achieve the same performance benefits as Duran's scheme. As our results show, even for these benchmarks, Lazy Scheduling was able to schedule the remaining parallelism more efficiently than some of the competing schedulers, without falling behind (on average) compared to the others. It is important to note, however, that parallelism

cut-off is only applicable to certain programs.

4.10 Analytical Comparison with other Work Stealers: A Second Approach

In this section, we revisit the question of time and space bounds for the different work stealing schedulers that we discussed. As mentioned, the nice theoretical bounds that were shown in [16] apply to computations with parallel function calls but not parallel loops or other constructs that introduce multiple tasks at once. The bounds rely on the fact that tasks are created one at a time and need to be amended for parallel loops. We start with the space bounds for different variants of work stealing, then discuss the time bounds. We conclude this chapter by manufacturing the worst possible scenario for lazy scheduling and discussing why it might not be an issue in practice.

4.10.1 Space Bounds

Remember that the space bound for vanilla work stealing (work stealing without parallel loops) is $P \cdot S_1$, where P is the number of workers and S_1 the space needed by the sequential (depth-first) execution. It is important to note that the bound is a function of the sequential space as we proceed to generalize the result in the presence of parallel loops.

Algorithm 4.3 Generic Parallel Loop

1: **for all** $i \in \{1, \ldots, N\}$ **do**

2: CODE(i)

3: **end for**

Let us assume a generic parallel loop with N iterations (tasks), such as the one shown in Algorithm 4.3. Work-first work stealing creates a single task-descriptor per loop and therefore the $P \cdot S_1$ bound still holds, ignoring $O(1)$ terms. Help-first

Scheduler	Space	Time
Work-First	$P \cdot S_1$	$T_1/P + T_\infty + \mathrm{O}(N)$
Help-First	$P \cdot S_1 + \mathrm{O}(N)$	$T_1/P + T_\infty + \mathrm{O}(N)$
Simple-Partitioner	$P \cdot S_1 + \mathrm{O}(\log N)$	$T_1/P + T_\infty + \mathrm{O}(\log N)$
Auto-Partitioner	$P \cdot S_1 + \mathrm{O}(\log K \cdot P)$	$T_1/P + T_\infty + \mathrm{O}(\log K \cdot P)$ *
Lazy Scheduling (BF-LS)	$P \cdot S_1$	$T_1/P + T_\infty + \mathrm{O}(\log P)$ *
Lazy Scheduling + XMT	$P \cdot S_1$	$T_1/P + T_\infty$ *

Table 4.31: Space and Time Bounds for Generic Parallel Loop

work stealing starts by creating N task descriptors, one for each iteration. Thus, the space needed is $P \cdot S_1 + \mathrm{O}(N)$. Eager binary splitting with simple-partitioner will recursively split the iteration range creating $\log N$ task descriptors, so the space will be $P \cdot S_1 + \mathrm{O}(\log N)$. Auto-partitioner will only create $K \cdot P$ chunks initially, so the space requirement is $P \cdot S_1 + \mathrm{O}(\log K \cdot P)$. Finally, all the lazy scheduling approaches have a constant bound on the number of task-descriptors in the work-pool (e.g., one task descriptor per deque), therefore the original bound of $P \cdot S_1$ holds. Table 4.31 summarizes these results.

4.10.2 Time Bounds

The time bound for vanilla work stealing is $T_1/P + \mathrm{O}(T_\infty)$, where T_1 is the work, i.e, the time taken by sequential execution of the parallel code, and T_∞ is the depth, i.e., length of the critical path. For the generic loop of Algorithm 4.3, for example, T_∞ is the length of the longest of its N tasks.

Table 4.31 shows the time bounds as well. Work-first work stealing removes the N tasks from the single task descriptor sequentially, so the time bound has an additional $\mathrm{O}(N)$ term. Help-first work stealing starts by creating N task descriptors before even executing the first task, so it also incurs an added $\mathrm{O}(N)$ on the critical path. Simple-partitioner has a logarithmic overhead for the same reason.

Auto-partitioner also has a logarithmic overhead but in $K \cdot P$ instead of N. Lazy Scheduling stops creating task descriptors as soon as thefts stop and the deque size grows above the threshold, in other words, after $\mathrm{O}(\log P)$ time, to account for the parallel distribution of task descriptors to all P workers in $\log P$ rounds. On XMT, because the hardware scheduler reduces these $\log P$ steps to a constant overhead (assuming the outer parallelism is at least P), this logarithmic overhead on the critical path is removed from the time-bound of Lazy Scheduling. This occurs because, in the *intermediate* and *best* cases (see Section 4.3), Lazy Scheduling drastically reduces the splitting that enters the critical path for other schedulers. Conversely, other schedulers, even with the XMT hardware scheduling of outer parallelism, do not shed their linear and logarithmic overheads from their time bounds because they do not reduce their overheads in the absence of thefts.

Overall, lazy scheduling has the best bounds, both for space and time, and the eager binary splitting approaches (SP and AP), are the next best.

Note that the time bounds for auto-partitioner and Lazy Scheduling hold only if the work of all tasks in the parallel loop is of the same order of magnitude. In the worst case, if an adversary picked the computational cost of each task in the loop, the time bounds for auto-partitioner and Lazy Scheduler degrade, as discussed in the next section.

4.10.3 Adversarial Scenarios for AP and Lazy Scheduling

Imagine that for auto-partitioner the first chunk of tasks of a parallel loop contains $\mathrm{O}(W)$ work per task, whereas all of the other tasks have practically no work, i.e., $\mathrm{O}(1)$. The critical path will be $T_\infty = W$, and because the first chunk will contain N/KP iterations, the total work will be:

$$T_1 = W \cdot \frac{N}{KP} + \left(N - \frac{N}{KP}\right) = (W + KP - 1) \cdot \frac{N}{KP}$$

However, since auto-partitioner will execute all N/KP tasks as a single chunk,

its critical path will be $T_\infty = W \cdot N/KP$, which is N/KP times longer than the critical path of the given code.

For Lazy Scheduling, a similar scenario induces a bad behavior. This time, assume that the first $\log N$ tasks have $\mathrm{O}(W)$ work, and the rest have $\mathrm{O}(1)$. Also, assume that we stop pushing work on a deque if it is not empty (the size threshold is one) and that thefts happen slower than checking the deque size after pushing a task descriptor. Lazy Scheduling will push a task descriptor with half the tasks onto its deque and will start executing the first of the "thick" $\mathrm{O}(W)$ tasks. In the meantime, all of the $N/2$ thin $\mathrm{O}(1)$ tasks are split among the remaining workers and executed very quickly. After executing its thick task, the worker that initiated the parallel loop will find its deque empty, push a task descriptor with half of the remaining tasks and start executing the next thick task. Once again, the $N/4$ thin tasks are split and executed by the remaining workers, and so on for $\log N$ rounds. Therefore, the worker that initiated the parallel loop will execute $\mathrm{O}(W) \log N$ work, which is $\log N$ times longer than the critical path of the original parallel loop.

Lazy Scheduling is not as bad as auto-partitioner in these scenarios, since its critical path increases by a logarithmic factor, instead of a linear one.

In most realistic scenarios, it is hard to imagine that forking off half of the tasks will fail to create enough work while the originating worker executes the first task, especially under the commonly used *slackness assumption* that the parallel tasks greatly outnumber the available workers ($T_1/T_\infty >> P$). From our experience, this is a reasonable assumption to make for task-parallel codes, even for non-declarative ones, and we have not encountered a situation that exposed the above vulnerability of Lazy Scheduling.

Nonetheless, we have some thoughts on how to overcome this drawback of Lazy Scheduling. An observation we have made is that declarative codes tend to have short tasks. In fact, all of our declarative benchmarks had $\mathrm{O}(1)$ work per task. Moreover, all the non-declarative ones would also have $\mathrm{O}(1)$ work per task, had we

made them declarative. In such codes, the critical path is the same as for vanilla work stealing, plus the unavoidable $\log P$ overhead to distribute the work. Another idea that could lead to a solution is to randomly choose which of the two halves of a task-descriptor to push on the deque after splitting it. While this does not improve the worst case, it may reduce the expected value of the critical path. Moreover, increasing the deque size threshold to k may reduce the factor by which the critical path increases to $\frac{\log N}{k}$, under the conservative assumption that thefts only happen during the execution of a thick task, and not during recursive splitting, which would be more convenient. Finally, if we modify the Lazy Scheduling to check the deque at roughly constant ($O(1)$) time intervals rather than only at task boundaries, the deque can be kept full and the critical path will likely be optimal, as the one shown in Table 4.31. The checks can possibly be injected by a compiler pass, or they can be triggered by a recurring hardware interrupt. The second idea might actually remove the need for coarsening to amortize overheads altogether by controlling how often deque checks are performed, as long as the interrupt is light weight and does not require a context switch. If such a mechanism can be implemented, then perhaps it is also worth considering to replace the recurring interrupt by an interrupt induced by a thief discovering an empty deque.

4.11 The Inception of Lazy Scheduling: an Interesting Anecdote

To close this section, I would like to present an anecdote as to how the idea for Lazy Scheduling, a central contribution of this work, came to be, and how XMT may have played a role in enabling the development of that idea. I had just finished implementing a basic XMTC compiler without support for nested parallelism, and I was starting work on nested parallelism. When it came to the implementation of the scheduler, I started implementing the popular work stealing algorithm. The size of its work-pool is unbounded, but parallel dynamic memory allocation was not available on XMT (since I had not implemented it); besides, using expensive dy-

namic memory allocation for scheduling fine-grained tasks seemed counter intuitive. So, I made a slight change to the scheduler to check if it could add a task to the local deque without overflowing it; if not, it would start executing a task and check again later instead of exiting. I set capacity of the local deques to something large, for example the number of worker threads P, so the total number of tasks in all the deques would be at most P^2.

When I started experimenting with fine-grained parallel benchmarks, my intuition was that pushing and popping tasks off the local deque is a significant waste if they end up executing locally. So, I started reducing the capacity of the deques and found that, with a capacity of a single task, the performance was maximized. That observation led to the realization that the size of the local deque could be used as a relatively accurate low-overhead approximation of the system load, and that led to the inception of Lazy Scheduling. However, with a capacity that small, without keeping track of *postponed* tasks (tasks that would have been added to the deque had its capacity been larger) the scheduling order of work stealing was violated, and the number of thefts (lightweight context switches) could increase dramatically, thus destroying (temporal) locality and, with it, performance. Luckily, XMT's design absolves programmers from most locality considerations, and the issue did not present itself, allowing me to attack one issue at a time. Later, I implemented the additional bookkeeping required to restore the desirable scheduling order for Lazy Scheduling and showed its value not only on XMT, but on commercial multicores as well. In conclusion, XMT offered a good platform for me to develop my ideas and bring them to maturity because it allows the programmer to attack a problem in waves: first, without care for locality, and then, by refining the solution to account for locality.

Chapter 5

The XMTC Compiler

In this chapter, we present information on the XMTC compiler and language. We start by presenting the XMTC Memory Model and the issues we encountered when using GCC, the popular GNU compiler for C and other sequential languages, as the basis for XMTC, a parallel language. These topics, along with some information on XMT specific optimizations were presented in [54]. Then, we proceed to give some more details on how outer spawn statements (i.e., parallel loops) are compiled to take advantage of XMT's hardware scheduling mechanisms and how we incremented this basic compiler to support nested parallelism.

5.1 Overview

The XMTC compiler translates XMTC code to an optimized XMT executable. Roughly speaking, the XMTC compiler consists of three consecutive passes: the *pre-pass* performs source-to-source (XMTC-to-XMTC) transformations and is based on CIL [70]; the *core-pass* performs the bulk of the compilation and is based on GCC v4.0; and the *post-pass*, built using SableCC [39], takes the assembly produced by the core-pass, verifies that it complies with XMT semantics and performs linking. The *xmtcc* bash script links the passes together for convenience and accepts multiple XMTC files, data files, and libraries as arguments.

One unusual aspect of the compilation of XMTC code is that assembling (converting assembly to binary) happens after linking. This choice was made to allow targeting both the XMT FPGA hardware [91, 92], which accepts binary executables, and the XMT cycle-accurate simulator [54], which accepts a big monolithic assembly file. This choice was made to make it easier to use the simulator as a debugging tool

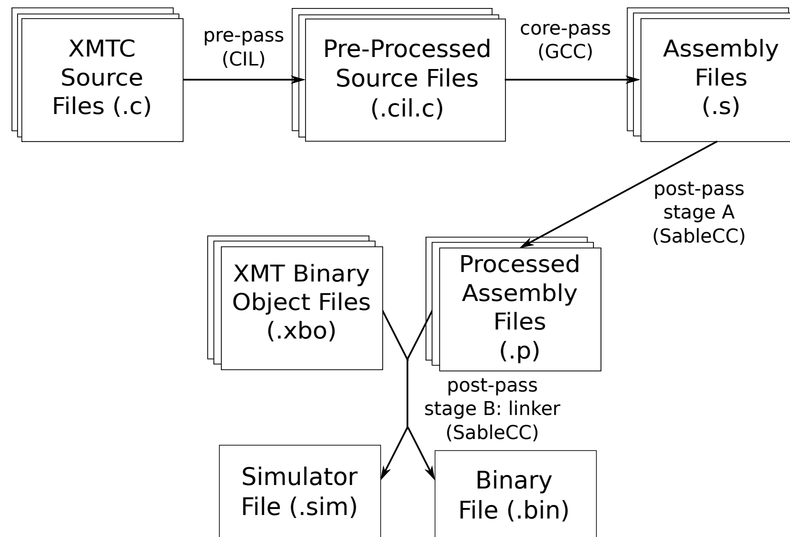


Figure 5.1: Compiler Passes.

and to allow adding new instructions in the simulator for experimentation, without having to define a binary representation for them (opcodes, instruction formats, etc).

Figure 5.1 gives an overview of the compilation process of an XMTC file. We kept the `.c` file extension to have text editors interpret and highlight the code as C code, since XMTC is an extension of C. The pre-pass takes one or more XMTC source files and produces an intermediate `.cil.c` file for each source file. Then, the core-pass produces an assembly (`.s`) file for each of these intermediate files. Finally, the post-pass first performs some transformations on the assembly producing a `.p` file for each `.s` file, then links all the files, possibly including library code and binary files (`.xbo`), and produces two files: a `.sim` file and a `.bin` file. The `.sim` file contains the instructions of the program in assembly and is used only when running the program on the simulator. The `.bin` file has different contents depending on whether the target platform of the compilation was the FPGA or the simulator. In the former case, the `.bin` file contains contains both the code and the initial data in binary format; in the latter case, it only contains the initial data in binary format, and the instructions are not included (they are replaced by zeros to maintain correct

addressing of data).

The different passes of the compiler host different functionalities of the compilation process. Here, we briefly list the functionality that each of the three compiler passes implement and we expand on them later in this chapter.

The pre-pass (CIL) is home to source-to-source transformations, including: *(1) function cloning*[1] to keep stack management in sequential code optimal and to keep track of which spawn-statements are nested; *(2) outlining of outer spawn statements* to prevent illegal data-flow across spawn-statement boundaries; *(3)* spawn-block and spawn-statement *cost prediction* and *automatic coarsening* based on that cost-prediction, including picking a granularity parameter for spawn-statements and static or dynamic serialization (see Section 3.5.1); *(4) flattening* of perfectly nested spawns to reduce the nesting depth and avoid scheduling overheads; *(5) outlining and conversion of nested spawn-blocks* to create closures for tasks that the scheduler will be called to execute.

CIL had to be modified to allow using it for XMTC. First, the lexer and parser had to be extended to include the XMTC extensions, but also the concrete syntax tree produced by the parser to include a new type of statement nodes, spawn-statements. Another much trickier modification that was required had to do with CIL's design to hoist all variable declarations within a function to the top of that function. While correct for a sequential C program, the XMTC spawn-statement implies that the scope of a variable declared within the spawn-block are private to it. In particular, multiple instances of the variable may exist simultaneously, as many as the tasks created by the spawn-statement. For that reason, hoisting of variable declarations is illegal in XMTC, and the internal data-structures of CIL had to be modified to support local variable declarations for each block of code. Finally, the procedure that builds the concrete parse tree in CIL had to be modified to correctly

[1]Function Cloning simply creates a parallel clone of each function in the code and updates each call-site to invoke the appropriate clone.

account for local variable declarations.

The core-pass (GCC) converts the intermediate source code to assembly. In addition to all the conventional GCC optimization passes, the core-pass implements: *(1) live-register broadcasting* for transitioning from sequential to parallel mode; *(2) cactus-stack allocation* to support a parallel stack for the parallel portions of the code; *(3) global register* loading and reading; as well as *(4) linear and loop prefetching* implemented by George C. Caragea [20].

GCC's parser had to be modified to parse XMTC, but its internal data structures were not. Instead, the new constructs were expressed using GCC's existing data structures. This choice was made to allow the use of all of GCC's optimizations without having to update them to account for the new types of statements and in particular explicit parallelism. For example, a spawn-statement is compiled as a `spawn` inlined assembly instruction (which GCC does not need to know about) that is in turn followed by the spawn-block, followed by a `join` inlined assembly instruction. This, of course, opens the door for illegal data-flow and code-motion, which are prevented by compiler passes, such as *outlining* (see Section 5.3). Furthermore, GCC's MIPS machine-description in GCC's backend was cloned and incremented to create XMT's machine-description and to describe the existence of global-register, as well as rules for managing them. Additionally, XMT specific passes were added in this backend, such as live-register broadcasting and cactus-stack allocation.

The post-pass starts by performing a battery of simple transformations on the assembly files produced by the core-pass, mainly straightforward assembly sanity checks, rewriting, and simplifications. Then, it applies some more involved transformations, namely *function insertion*, *dead function elimination*, *assembly block reordering*, *burst prefetching*, *global address and label calculation*, *linking of data-files*, and *assembling*.

The post-pass was built from scratch using the SableCC parser generator. The grammar describing the language for parsing the assembly produced by the core-pass

is an extended version of the grammar written by Fuat Keceli for his cycle-accurate XMT simulator [54, 52].

5.2 The XMTC Memory Model

The memory consistency model for a parallel computing environment is a contract between the programmer and the platform, specifying how memory actions (reads and writes) in a program appear to execute to the programmer, and specifically which values reading a memory location may return [64].

Consider the example in Figure 5.2. If memory store operations are non-blocking, meaning they do not wait for a confirmation that the operation completed, it is possible for Task B to read {x=0 and y=1}. At first this *relaxed consistency* is counter-intuitive, but because it allows for much better performance by allowing multiple pending write operations, it is usually favored over more intuitive but also more restrictive models [4, 64].

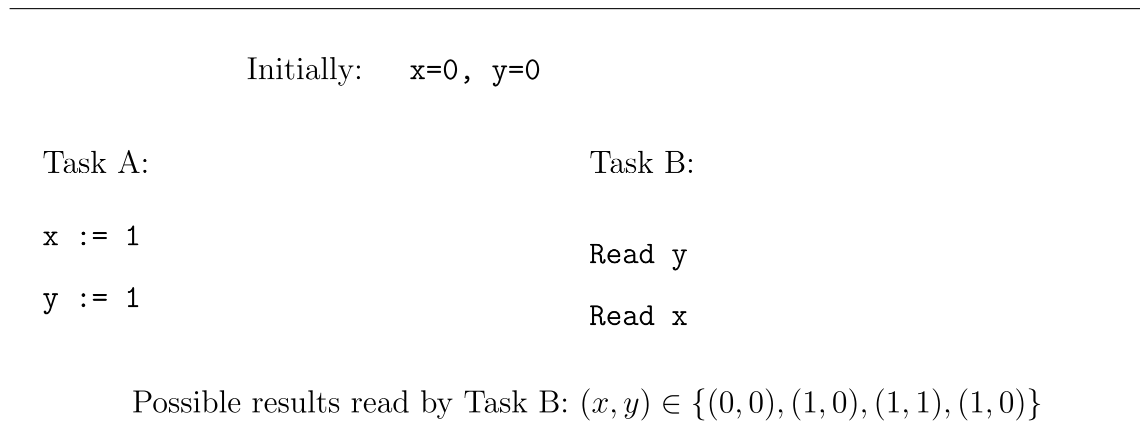


Figure 5.2: Two tasks with no order-enforcing operations or guarantees.

The XMT memory model is a relaxed model that allows the same results for Task B as in the previous example. It relaxes the order of memory operations and only preserves relative ordering with respect to prefix-sum operations (`ps` and `psm`), and to the beginning and end of `spawn` statements. This makes prefix-sum operations important for synchronizing between tasks, as will be shown in Figure 5.3.

The XMT memory model gives the programmer two rules about the ordering of (read and write) memory operations. *First, it guarantees sequential execution within one task*, which means that a read to a memory location will return the last value written to that location by the current task, provided it was not overwritten by a different task. Intuitively, read or write operations from the same source (TCU) to the same destination (memory address) will not be reordered, neither by the hardware, nor by the compiler. This rule allows the programmer to treat each task as they would treat sequential code, as long as the tasks do not modify shared data (i.e., there are no data-races). The next rule deals with disambiguating (synchronizing) access and modification of shared data.

Second, for each pair of tasks, the XMT memory model guarantees a partial ordering of memory operations relative to prefix-sum operations over the same base. This rule allows the programmer to reason about *"happens before"* [58] relations and to enforce synchronization between concurrent tasks.

This rule is a bit more involved, so we explain it through the example in Figure 5.3. This example shows how to implement the example of Figure 5.2 if we want the invariant *if y=1 then x=1* to hold at the end of Task B (*i.e.*, disallow $(x, y) = (0, 1)$). Both tasks synchronize (in a loose sense) over variable `y` using a `psm` operation; task A writes (increments) `y` whereas task B reads it. At run-time, one of the two tasks executes its `psm` instruction first; the second rule of the XMT memory model guarantees that all memory operations issued before the `psm` of the first task to execute its `psm` will have completed before any memory operation after the `psm` of the second task is issued. In our example, assume that task A completes its prefix-sum first. That means that the operation `x=1` completed before task B reads X, which enforces the desired invariant *if y=1 then x=1* for task B.

The implementation of these two rules by the hardware and the compiler is straightforward. For the first rule, the static hardware routing of messages from TCUs to memory guarantees that the order of operations issued from the same

Initially: `x=0, y=0`

Task A:

```
x = 1;
tmpA = 1;
psm(tmpA, y);  //y++
```

Task B:

```
tmpB = 0;
psm(tmpB, y); // Read y
.. = x        // Read x
```

Possible results read by Task B: $(x, y) \in \{(0,0),(1,0),(1,1)\}$

Figure 5.3: Enforcing partial order in the XMT memory model.

source to the same destination will be preserved. The compiler enforces the second rule by (a) issuing a *memory fence* operation before each prefix-sum operation to wait for all pending writes to complete, and by (b) not moving memory operations across prefix-sum instructions. The current implementation does not take into account the base of prefix-sum operations and may be overly conservative in some cases. Using static analysis to reduce the number of memory fences and to selectively allow motion of memory operations across prefix-sums could be the topic of future research. It is unlikely, however, that such an optimization would yield substantial benefits since prefix-sum operations are typically used to synchronize between tasks. In the future, if more types of atomic operations are added to XMTC, the memory model may have to be updated to enforce partial ordering with respect to them as well.

Note that in Figure 5.3 both `psm` operations are needed. If, for example, Task B used a simple read operation for `y` instead of a prefix-sum, prefetching could cause variable `x` to be read before `y` and the invariant *if y=1 then x=1* would not hold.

An implication of the XMT memory model is that register allocation for parallel code is performed as if the code were serial. The programmer, however, must still declare variables that may be modified by other tasks as `volatile`. These variables will not be register allocated, as in the case of multi-threaded C code. This is rarely needed in XMTC user code, but it is useful in low-level library and system code.

Finally, the memory-fence operation is available to expert programmers who want to explicitly enforce ordering of specific memory operations because sometimes a prefix-sum operation is not ideal. In the example above, task A would simply set `x` to 1, invoke the memory-fence, then set `y` to 1. This would have the same effect as the code of Figure 5.3 but with the added benefit that `y` would be written using a non-blocking store operation, which is more efficient than the blocking `psm` operation.

5.3 Compiling XMTC Parallel Code with a Serial Compiler

Although we have extended the core-pass (GCC) to parse the additional XMTC parallel constructs, it inherently remains a compiler for sequential C. Changing the internal GCC data structures to express parallelism would have required great effort, and all the optimization passes would have had to be updated to account for these new constructs. Such a task was beyond the scope of this work and fortunately proved unnecessary in practice. Instead, a spawn statement is parsed as if there was a `spawn` inlined assembly instruction at the beginning of the spawn block and a `join` at the end of it. Figure 5.4(a) shows the code as written by the programmer, whereas Figure 5.4(b) shows how the compiler interprets it. Therefore, GCC interprets a spawn statement as a sequential block of. This opens the door for illegal dataflow because *(1)* it hides the fact that the spawn block might be executed multiple times (i.e., its loop semantics), *(2)* it hides the concurrency of these multiple executions, and *(3)* it hides the transfer of control from the Master TCU to the parallel TCUs where the spawn-block is executed in the case of an outer spawn statement.

An invalid code transformation caused by illegal dataflow is code-motion across spawn-block boundaries. For example, the code of Figure 5.4(a) reads all the elements of array A in parallel and, if an element is non-zero, it sets `found` to `true`. After the parallel section, `counter` is incremented if a non-zero element was found.

The compiler may choose to move the conditional increment statement `if(found) counter+=1` before the `join` instruction to issue the non-blocking store operation for `counter` earlier and overlap it with the `join` instruction. The counter could then be incremented multiple times instead of only once, which breaks the semantics of the original program.

5.3.1 Outlining

To prevent illegal dataflow, we implemented *outlining* (also known as *method extraction*) in the CIL pre-pass, an operation akin to the reverse of function inlining. Figure 5.4(c) shows the outlined version of the code in Figure 5.4(a). Outlining places each spawn statement in a new function and replaces it by a call to this new function. According to XMTC semantics, the spawn statement should have access to the variables in the scope of the enclosing serial section, so the outlining pass detects which of these variables are accessed in the parallel code and whether they might be written to. Then, it passes them as arguments to the outlined function by value or "by reference" accordingly. In Figure 5.4(c), because the variable `found` is updated in the spawn block, a pointer to it is passed to the outlined function, and the spawn block is updated to access it through the pointer.

Outlining prevents illegal dataflow without requiring all optimizations to be turned off. This solution works because GCC, like many compilers, does not perform inter-procedural optimizations. Compilers that do perform inter-procedural optimizations often provide a flag that has the effect of preventing inter-procedural code motion.

5.3.2 Register Broadcasting

We now present another example of illegal dataflow, but this time without code motion. It happens because GCC is unaware of the transfer of control from the serial processor (Master TCU) to the parallel TCUs that a spawn statement

(a) Original Code

```
int A[N];
bool found=false;
spawn(0,N−1) {
    if (A[$]!=0)
        found = true;
}
if (found) counter+=1;
```

(b) What the compiler sees

```
int A[N];
bool found=false;
asm(spawn 0, N−1);
if (A[$]!=0)
    found = true;
asm(join);
if (found) counter+=1;
```

(c) After Outlining

```
int A[N];
bool found=false;
outl_sp_1 (A, &found);
if (found) counter+=1;
....................

void outl_sp_1(int (*A),
               bool *found) {
    spawn(0,N−1) {
        if (A[$]!=0)
            (*found) = true;
    }
}
```

Figure 5.4: Simple example of outlining.

entails on XMT. GCC optimizations incorrectly assume that a value can be loaded to a register before the spawn statement (within the outlined function) and later accessed within the spawn-block. This is not the case because the value is loaded to a Master TCU register while the spawn-block code accesses the TCU registers. There are two ways to fix this problem: (a) move the load instruction back into the spawn-block, causing each task to load the value from memory, potentially creating a memory hot-spot, or (b) broadcast all live Master TCU registers (an XMT specific operation) to the parallel TCUs at the onset of a task. We chose the second approach because it conserves memory bandwidth.

(a) Wrong layout by GCC	(b) Corrected layout

<pre>outlined_spawn:
    spawn
BB1:
    ...
    bneq $r, $0, BB2
    join
    jr $31 # return
BB2:
    ...
    j BB1</pre>
	<pre>outlined_spawn:
	    spawn
	BB1:
	    ...
	    bneq $r, $0, BB2
	    j BBjoin // GCC tried to
	             //save this jump
	BB2:
	    ...
	    j BB1
	BBjoin:
	    join
	    jr $31 # return</pre>

Figure 5.5: Example of assembly basic-block layout issue.

5.3.3 Assembly Code Layout Correction

Finally, XMT places a restriction on the layout of the assembly code of outer spawn blocks, because it needs to broadcast the code to the TCUs. The restriction is that all spawn-block code must be placed between the `spawn` and `join` assembly instructions. Interestingly, in its effort to optimize the assembly, GCC might decide to place a basic-block (a short sequence of assembly instructions) that logically belongs to a spawn-block after it. In the example of Figure 5.5(a), basic-block 2 (BB2) is placed after the return statement of the `outlined_spawn` function to save one jump instruction.

The assembly code produced by GCC has correct semantics, but it will lead to incorrect execution on XMT because BB2 will not be broadcast by the XMT hardware, and TCUs do not currently have access to instructions that were not broadcast. Future versions of XMT will allow TCUs to fetch instructions that are

not in their instruction buffer by including instruction caches at the cluster or TCU level.

One way to avoid this code layout bug would be to disable the offending optimization passes, but that would prevent the optimizations from happening even when they are legal. Instead, the post-pass checks for layout violations and fixes them by relocating misplaced basic-blocks, as shown in Figure 5.5(b).

5.3.4 Why illegal dataflow is not an issue for thread libraries

There are several libraries that are used to introduce parallelism to serial languages (e.g., *Pthreads*). Code written using such library calls are compiled using a serial compiler, so one might wonder why illegal dataflow is not an issue in that scenario. In Pthreads, the programmer creates an additional thread using the `pthread_create` call, which takes as an argument a function to execute in the new thread. In other words, the programmer is forced to do the outlining manually. Moreover, thread libraries do not introduce new control structures in the base language, such as XMTC's spawn statement, so the compiler does not need to be updated. That said, serial compilers can still perform illegal optimizations on Pthreads code [17], but these are rare enough so that Pthreads can still be used in practice. The main disadvantages of using a thread library, however, is the lack of compiler optimizations specifically targeting parallel code and the added complexity for the programmer: creating parallelism through a library API is arguably harder and less intuitive than directly creating it using parallel constructs incorporated in the programming language. The converse argument, that libraries are preferable to new languages, has also been stated [89], primarily on the basis of backward compatibility with sequential code-bases and of the reluctance of programmers to learn new languages. We believe that parallel extensions to existing languages are preferable because they provide backwards compatibility, cleaner and easier parallel coding, and better performance. We concede, however, that extending a programming lan-

guage is a much more substantial effort than writing a library, but we believe it is worth the effort.

5.4 XMT-Specific Optimizations

Some of the design decisions of XMT create new opportunities for compiler optimizations that were not possible or not needed for other architectures. Novel parallel architectures may adopt similar designs, making these optimizations relevant to them. This section presents some of them.

5.4.1 Latency tolerating mechanisms.

The XMT memory hierarchy is designed to allow for scalability and performance. To avoid costly cache coherence mechanisms (in terms of chip resources as well as performance overheads), the first level of cache is shared by all the TCUs, with an access latency in the order of 30 clock cycles for a 1024 TCU XMT configuration. Several mechanisms are included in the XMT architecture to overlap shared memory requests with computation or to avoid them: non-blocking stores, TCU-level prefetch buffers, and cluster-level read-only caches. Compiler managed scratch-pad memory per TCU or per cluster is also on XMT's roadmap.

Currently, the XMT compiler includes support for automatically replacing eligible writes with non-blocking stores and for inserting prefetching instructions to fetch data in the TCU prefetch buffers. Support for automatically taking advantage of the read-only caches is planned for future revisions of the compiler. In the meantime, programmers can explicitly load data into the read-only caches if needed.

The XMT compiler offers three prefetching options: linear prefetching, loop prefetching, and burst prefetching. The first two were developed by George C. Caragea as part of his dissertation [21], whereas burst prefetching was contributed by myself, as it helps to efficiently support function calls in parallel mode.

Linear prefetching issues a prefetch instruction for each memory load and the modified instruction scheduler pass in GCC tries to hoist the prefetch instruction up the control-flow graph (CFG) to hide as much latency as possible. The lack of local caches on XMT makes linear prefetching very profitable as it compensates for the lack of spatial locality normally provided by caches. Linear prefetching is not resource-aware, and in some cases, it can degrade performance by thrashing the prefech buffer with requests that overwrite one another. This happens, for example, when more prefetches than prefetch buffer locations are issued simultaneously.

The loop prefetching mechanism was designed to match the characteristics of a lightweight, highly parallel many-core architecture. It has been shown to outperform state-of-the-art prefetching algorithms such as the one included in the GCC compiler, as well as hardware prefetching schemes [20]. The key observation is that taking into account the size of the prefetch buffer and reducing the prefetch distance accordingly benefits performance. The prefetching algorithm is potentially applicable to other many-core platforms with small prefetch buffers.

5.4.1.1 Burst Prefetching

Burst prefetching hides latency when more than one contiguous memory load instruction is encountered. This is common in RISC architectures such as XMT, especially when a function returns, and it has to restore the *callee saved* registers it modified. Load operations are blocking, so a sequence of N loads requires N round-trips to memory, but this number can be reduced by means of prefetching. Linear prefetching misses this opportunity because it is implemented in the front-end of the compiler, before register allocation and stack management code generation. On the other hand, burst prefetching is implemented in the post-pass with the specific goal to act after those compiler passes produce assembly code. Like linear prefetching, burst prefetching is useful because of the lack of local caches that would provide spatial locality.

(a) Assembly Produced by GCC	(b) After Burst Prefetching
	`pref 8, 44($sp)   // 1`
	`pref 8, 40($sp)   // 1`
	`pref 8, 36($sp)   // 1`
	`pref 8, 32($sp)   // 1`
`lw $31, 48($sp)`	`lw $31, 48($sp)   // RTTM`
`lw $23, 44($sp)`	`lw $23, 44($sp)   // 1`
`lw $22, 40($sp)`	`lw $22, 40($sp)   // 1`
`lw $21, 36($sp)`	`lw $21, 36($sp)   // 1`
`lw $20, 32($sp)`	`lw $20, 32($sp)   // 1`
5 Round-Trips to Memory(RTTM)	1 RTTM + 8

Figure 5.6: Simple Burst Prefetching Example.

Figure 5.7(a) shows a snippet of the assembly code produced by the core-pass (GCC) when compiling the *QUEENS* benchmark. Nine consecutive load instructions are issued to restore registers before returning from a function, which results in nine round-trips to memory (RTTM). Figure 5.7(b) shows the same code after the burst prefetching has inserted prefetching instructions. Burst prefetching is resource aware, taking the size of the prefetch buffer as an input, but does not try to hoist the inserted prefetches after they are inserted. The key idea is to use a prefetch-buffer miss (similar to a cache miss) to hide the latency of multiple prefetches. To illustrate that concept we use a simpler example.

Figure 5.6(a) shows five consecutive load instructions resulting in five RTTM. Assuming the prefetch buffer can hold four words (elements), burst prefetching will issue prefetch instructions for the last four load instructions as shown in Figure 5.6(b). The first load instruction will incur a round trip to memory since it was not prefetched (prefetch-buffer miss), but, during that time, the four issued prefetches will presumably have time to complete. In reality, queuing at various points in the system may cause them not to complete, but they will have made

(a) Assembly Produced by GCC	(b) After Burst Prefetching
	`pref 8, 44($sp)` // *1*
	`pref 8, 40($sp)` // *1*
	`pref 8, 36($sp)` // *1*
	`pref 8, 32($sp)` // *1*
`lw $31, 48($sp)`	`lw $31, 48($sp)` // *RTTM*
`lw $23, 44($sp)`	`lw $23, 44($sp)` // *1*
	`pref 8, 28($sp)` // *1*
`lw $22, 40($sp)`	`lw $22, 40($sp)` // *1*
	`pref 8, 24($sp)` // *1*
`lw $21, 36($sp)`	`lw $21, 36($sp)` // *1*
	`pref 8, 20($sp)` // *1*
`lw $20, 32($sp)`	`lw $20, 32($sp)` // *1*
	`pref 8, 16($sp)` // *1*
`lw $19, 28($sp)`	`lw $19, 28($sp)` // *RTTM–6*
`lw $18, 24($sp)`	`lw $18, 24($sp)` // *1*
`lw $17, 20($sp)`	`lw $17, 20($sp)` // *1*
`lw $16, 16($sp)`	`lw $16, 16($sp)` // *1*
9 Round-Trips to Memory (RTTM)	2 RTTM + 9

Figure 5.7: Complete Burst Prefetching Example.

substantial progress nonetheless. Overall, the code after burst prefetching takes 1 RTTM + 8 cycles to complete, as opposed to 5 RTTM for the original code. This is substantial, considering that one RTTM measures around 25 cycles on the 64 TCU XMT FPGA and can take longer on XMT configurations with more TCUs. Note that the burst prefetching will issue prefetches for all the load instructions if they fit in the prefetch buffer: if there were 4 loads in the above example, they would all be prefetched.

The example in Figure 5.6 illustrates that with a prefetch buffer of K words, the number of RTTMs can be reduced by a factor of $K + 1$ and a small number of cycles will be added for issuing the prefetching instructions. Figure 5.7 shows what

Algorithm 5.1 Burst Prefetching Algorithm

1: INPUT: *instructions*: list of load instructions
2: OUPTUT: *output*: list of instructions including prefetching
3: **while** you have not considered all instructions for prefetching **do**
4: ▷ Go through the *instructions* list from head to tail
5: **if** The number of load instructions remaining is larger than the size of the Prefetch Buffer AND there is no upcoming load instruction that was not prefetched (that was skipped) **then**
6: Skip prefetching for the load instruction under consideration, keep track of the instruction (and consider the next load instruction in the next iteration)
7: **else**
8: Build a prefetch instruction for the load currently under consideration and add it to the *output*. Also increment the number pending prefetch instructions.
9: **while** The number of pending prefetches is $\geq$ than the Prefetch Buffer Size **do**
10: Move a load instruction from the head of *instructions* to the tail of *output*
11: If that load instruction was prefetched, decrement the number of pending prefetches, else (it was skipped) mark that there is no longer an upcoming load instruction that was not prefetched.
12: **end while**
13: **end if**
14: **end while**
15: Enqueue the rest of *instructions* to *output*

happens when the number of consecutive load instructions is larger than K (but smaller than $2K$). The first part is identical: four prefetches are issued; then, load instructions are inserted and as prefetch buffer locations are freed by them, more prefetches are issued.

Two rules form the core of the burst prefetching algorithm as it traverses the list of consecutive loads. First, if the number of remaining load instructions is greater than the size of the prefetch buffer, a load is *skipped* (no prefetch is issued for it) and the miss it will cause will allow the pending prefetches to complete. Second, prefetches are issued as soon as possible taking into account the size of the prefetch buffer. When pending prefetches fill up the prefetch buffer, load instructions are pushed to the output instruction list until a prefetch buffer location is freed (consumed).

Algorithm 5.1 gives a verbal description of the algorithm of burst prefetching and Algorithm 5.2 presents its pseudocode.

Closed Formula for Burst Prefetching. To define the closed form formula for the number of cycles taken by a sequence of lw instructions given a prefetch buffer of size S, we first define some values. Let L_{RTTM} be the number of load instructions that require a round trip to memory. Those are $L_{RTTM} = \lceil \frac{lw}{S+1} \rceil$. The number of remaining load instructions is $L_R = lw - L_{RTTM}$. The number of load instructions that were **not** prefetched is $N = \lfloor \frac{lw}{S+1} \rfloor$, and the number of prefetched instructions is $P = lw - N$. Let O be the number of overlapped cycles by prefetches issued before a load but without an intermediate RTTM operation. This number is convoluted and only reduces the cycle count slightly. Without it (assuming it is zero), we still get a pretty accurate upper bound on the number of cycles it takes to execute lw load instructions with burst prefetching, using a prefetch buffer of size S (see Equation 5.1).

$$Cycles(lw, S) = RTTM \cdot L_{RTTM} + L_R + P - O \tag{5.1}$$

To define the number of overlapped cycles O we need to define the following values: let $m = lw \bmod (S + 1)$; let a be zero if $m = 0$ and one otherwise ($a = \{0 \text{ if } m = 0, 1 \text{ if } m \neq 0\}$); let c be zero if $lw \leq S$ and one otherwise ($c = \{0 \text{ if } lw \leq S, 1 \text{ if } lw > S\}$). The number of overlapped cycles is then:

$$O = a \cdot ((m - 1) + c \cdot (S - 1)) \tag{5.2}$$

Algorithm 5.2 Burst Prefetching Pseudocode

```
1: procedure BurstPrefetching(instructions)          ▷ List of load instructions
2:     hasSkipped ← false               ▷ true when the next load was not prefetched
3:     prefCnt ← 0                                   ▷ Number of pending prefetches
4:     read ← instructions.head                         ▷ Pointer to read next load
5:     insert ← instructions.head                     ▷ Pointer to insert next load
6:     ouput ← ∅                                ▷ List of instructions with prefetches
7:                                  ▷ Invariant A: insert(.next)^(prefCount+s) = read,
8:                         ▷ where s = 1 if hasSkipped = true, and s = 0 otherwise.
9:     while read ≠ instructions.tail do
10:        if hasSkipped = false ∧ remaining loads > PrefBuff.size then
11:                                     ▷ Skip load instruction pointed to by read
12:            hasSkipped ← true
13:            skipped ← read
14:            read ← read.next
15:        else
16:            pref ← build prefetch instruction for read
17:            output.enqueue(pref)
18:            read ← read.next
19:            prefCount ← prefCount + 1
20:            while prefCnt ≥ PrefBuff.size do
21:                output.enqueue(insert)
22:                insert ← insert.next
23:                if hasSkipped = true ∧ skipped = insert then
24:                    hasSkipped ← false
25:                else
26:                    prefCnt ← prefCnt − 1
27:                end if
28:            end while
29:        end if
30:    end while                                        ▷ invariant: read = tail
31:    while insert ≠ tail do                         ▷ Invariant A stops holding
32:        output.enque(insert)
33:        insert ← insert.next
34:    end while
35:    return output
36: end procedure
```

5.5 Compiling a flat XMTC spawn statement

This section focuses on how a flat spawn statement, i.e., one without nested parallelism, is compiled down to assembly. The interesting aspect is how the XMT hardware is harnessed for scheduling such outer spawn statements. For simplicity, we show how outer spawns were compiled before nested parallelism was supported, and later in this chapter, we revisit this information to include support for nested parallelism.

(a) XMTC code	(b) after core-pass	(c) after post-pass

spawn(low, high)
{
  BlockCode($)
}
	spawn $rLow, $rHigh
	 bcast live regs
	 BlockAssembly($)
	join
		1 mvtg $grHigh, $rHigh
		2 getid $rTmp, $rLow
		3 mvtg $grLow, $rTmp
		4 spawn
		5 broadcast $rLow, $rLow
		6 getid $rId, $rLow
		7 spawn_start:
		8 chkid $rId, $grHigh
		9
		10 bcast live regs
		11 BlockAssembly($)
		12
		13 move $rId, 1
		14 ps $rId, $grLow
		15 jump spawn_start
		16 join

Figure 5.8: Compiling a flat spawn statement

Figure 5.8(a) shows a generic spawn statement. Its arguments, low and high, are expressions that evaluate to integers. The `BlockCode` is parametric in the task ID ($). In Figure 5.8(b), the core-pass (GCC) generates high level XMTC assembly: the `spawn` instruction has two integer register arguments that hold the values of the lowest and the highest task IDs to be executed; the `join` instruction marks the end of the task code; the `BlockAssembly` is a straightforward compilation of the `BlockCode`, before which the live Master TCU registers are broadcast to the corresponding TCU registers. The `spawn` and `join` instructions are further expanded by the post-pass to make use of XMT's capabilities of hardware scheduling and

synchronization as shown in Figure 5.8(c).

The XMT specific instructions used in Figure 5.8(c) to expand the `spawn` and `join` statements are explained below. Remember that each TCU and the Master TCU have their own private set of local registers and that global registers can be accessed and modified both by the Master TCU and by all parallel TCUs. Local registers are named $rX and global registers as $grX.

- **getid $rX, $rY:** When used in serial mode (i.e., by the Master TCU), it adds the value of register $rY to the number of TCUs in the system and stores the result in register $rX. When used in parallel mode, it adds the value of register $rY to the identification number of the TCU executing the instruction and stores the result in register $rX. TCUs are numbered from zero to $P-1$ on a P-TCU XMT.

- **mvtg $grX, $rY:** Moves the value of register $rY to global register $grX. This instruction is only valid in sequential mode (it can only be executed by the Master TCU).

- **broadcast $rX, $Y:** Copies the value of register $rY of the Master TCU to register $rX of the TCU executing the instruction. This instruction is only valid in parallel mode (it can only be executed by TCUs but not the Master TCU).

- **ps $rX, $grY:** Performs the hardware prefix-sum operation. It atomically adds the value of register $rX to global register $grY and sets $rX to the value of $grY before the operation. This instruction is only valid in parallel mode (it can only be executed by TCUs but not the Master TCU).

- **chkid $rX, $grY:** This instruction is only valid in parallel mode (it can only be executed by TCUs but not the Master TCU). It compares the value of register $rX to the value of global register $grY. If $\$rX \leq \$grY$ execution

proceeds to the next instruction; otherwise the TCU blocks until $\$rX \leq \$grY$, e.g., some other TCU increments $grY, or until all TCUs are blocked at a `chkid` instruction, signaling the end of the parallel section and the return to sequential execution on the Master TCU. This instruction implements a type of barrier and allows quick and efficient transition from parallel to sequential execution.

The expansion of the high level `spawn` and `join` instructions are shown in Lines 1-8 and 13-16 of Figure 5.8(c). Line 1 sets $grHigh to $rHigh, the ID of the last task to execute. Line 2 sets $rTmp to the sum of the number of TCUs and $rLow, the ID of the first task to execute ($\$rTemp = low + |TCUs|$). Line 3 sets the global register $grLow to the value of $rTemp. This is done because $grLow holds the id of the next available task, since TCUs will each initialize their task ID to $Task_{id} = low + TCU_{id}$ (Line 6), as soon as control is transfered to them with the `spawn` instruction (Line 4). Each time a TCU completes its task, it will atomically increment $grLow using a prefix sum operation (Line 14) and check if the task ID is valid,i.e., $\leq \$grHigh$ (Line 8). Parallel execution is the started by Line 4. Then, the value of $rLow is broadcast from the MTCU to the TCUs (Line 5) as it will be needed for the TCUs to compute their initial task IDs. Line 6 sets the task ID to $low + TCU_{id}$, as mentioned earlier, and Line 8 only allows TCUs with valid IDs to proceed while blocking all the others. The expansion of the join is straightforward: Line 13 sets an increment of 1 for the prefix-sum operation of Line 14, which returns the next available task-ID (and increments $grLow); then, the execution jumps to the `checkid` instruction to validate this newly acquired task-ID.

XMT's hardware implements a first-in first-out (FIFO) schedule of the tasks, akin to using a global queue for scheduling. While this is good for flat parallelism, it results in a potentially unbounded memory footprint in the presence of nested parallelism. An interesting question is how to best use XMT's hardware to support nested parallelism, while keeping the memory footprint bounded. As mentioned

previously, we use the hardware to schedule outer spawn statements and use software scheduling for nested spawn statements. More details on how this is achieved are provided in Section 5.6.7.

5.6 Nested Parallelism Support

Since we want to use XMT's hardware to schedule outer spawn statements we need to have a way to differentiate between the two. For example, although the `spawn` on Line 3 is certainly nested in Figure 5.9, we do not know for sure whether the one on Line 2 is nested or not. It depends on whether the function `foo` was called from a sequential or a parallel context (execution mode).

```
1  void foo() {
2    spawn (1, 100) { // Outer?
3      spawn (1, 50} { // Nested
4          ...
5      }
6  }
```

Figure 5.9: Identifying Outer and Nested Spawns

5.6.1 Function Cloning

Outer spawn statements can be identified statically after performing *function cloning*. A parallel clone of each function is created and all function call-sites updated to call the appropriate clone. The original function is called from sequential contexts, and the parallel clone is called from parallel contexts (Figure 5.10).

Function pointers are not currently supported in XMTC. To support them, the appropriate clone would have to be selected dynamically at run-time, unless the compiler could statically disambiguate which clone was needed. The mechanism for picking the right clone at compile time is similar to picking the appropriate version

(a) original code	(b) after function cloning

`void foo( ... ) {`
`  bar(); // function call`
`  spawn (low, high) {`
`    bar(); // function call`
`  }`
`}`
	`void foo( ... ) {`
	`  bar(); // function call`
	`  spawn (low, high) {`
	`    par_bar(); // fn call`
	`  }`
	`}`
	`void par_foo( ... ) {`
	`  par_bar(); // fn call`
	`  spawn (low, high) {`
	`    par_bar(); // fn call`
	`  }`
	`}`

Figure 5.10: Function Cloning

of a *virtual* function of an object based on its type in object oriented languages. We do not expect this to be difficult to resolve in the future as the XMT platform matures.

5.6.2 Function Call Support in Parallel Code: Stack Allocation

To support recursively nested parallelism, the programmer must be allowed to call a function from parallel code. To make this possible, we implemented parallel stack allocation. However, parallel stack allocation is trickier than simply maintaining a linear stack for each TCU. The reason is that according to the semantics of a spawn statement, its spawn block has access to the variables in the enclosing scope. For example, in Figure 5.11(a) the parameter N should be accessible within the spawn block, but because it is modified, a single copy of N is maintained. Multiple copies may be preferable in the case of variables that are only read by tasks. Figure 5.11(b) shows that tasks should have their own activation frame as they can be running on different workers, but they should have access to the frame of the parent

task, which in the example holds N. Such a tree of activation frames is called a *cactus-stack*.

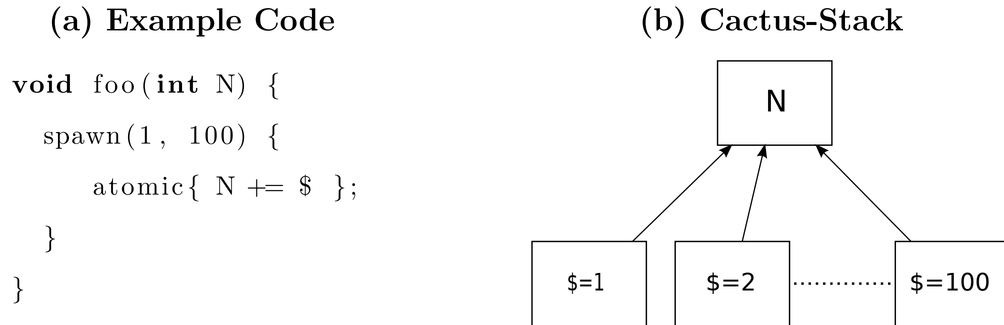


Figure 5.11: Spawn Scope Example

A cactus-stack can be implemented many different ways. An obvious way is to dynamically allocate each activation frame and keep a pointer to the parent frame. But when a task calls a function sequentially, this overhead is unnecessary, and something more akin to a linear stack could be used.

In the fourth chapter of his PhD thesis [40], Goldstein presents and compares four implementations of cactus-stacks: linked frames, spaghetti stacks, stacklets, and multiple stacks. The trade-offs of these implementations include allocation efficiency, internal fragmentation, external fragmentation, and ease of implementation. Internal fragmentation is caused by leaving unused space within the basic allocation unit and external fragmentation arises when allocating activation frames in different regions of the memory space.

Of the four alternatives, *stacklets*, proposed by Goldstein in his earlier work [41], achieved the best performance, especially for fine-grained parallelism. I chose to implement a cactus-stack based on Goldstein's *stacklets* on XMT, despite the fact that implementing them is relatively complex. Having a sequential and a parallel clone of each function minimizes stack allocation overheads for sequential parts of the code that do not need the additional complexity of cactus-stacks. This can also be achieved by compiling functions to have two entry points in assembly, one

which includes stacklet allocation and the other skipping it. In any case, stack is allocated as usual for the sequential clone of a function, and the more complex stacklet allocation code of the cactus-stack is only generated for the parallel clone.

The main motivation of stacklets is to reduce the stack allocation cost in the common event of a task calling a function sequentially, as opposed to spawning new tasks. In that case, the allocation will be almost as efficient as sequential stack allocation.

A stacklet is a continuous chunk of memory (2KB in the XMTC implementation) with some necessary information, such as a link to the parent stacklet. Activation frames are allocated within a stacklet as on a sequential stack, with just one additional check that the activation frame to be allocated fits in the remaining space in the stacklet; otherwise a new stacklet is allocated, linked to the current stacklet and the activation frame is allocated at the base of the new stacklet.

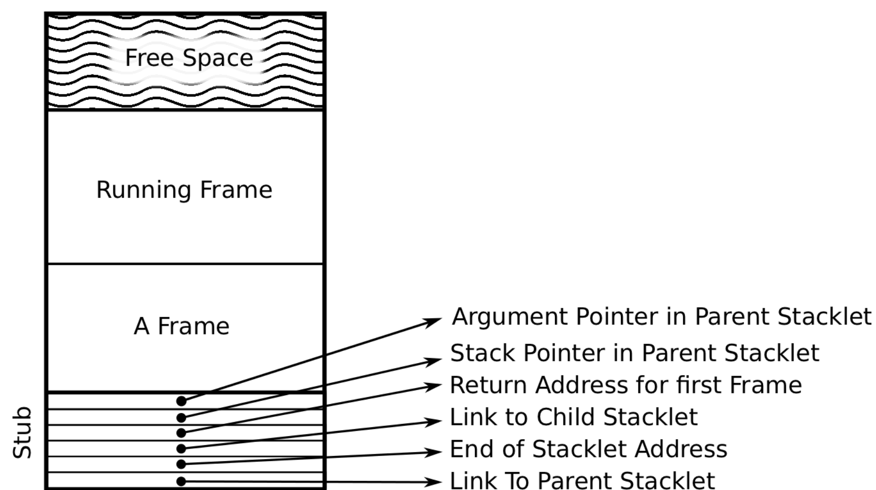


Figure 5.12: Stacklet

Figure 5.12 shows a stacklet. It has a stub, some activation frames and free space at its top. The activation frame at the top of the stacklet is the running frame, unless the stacklet has an active child. The stub stores some necessary information

to keep stacklets linked together and to restore the state of a function returning to the parent stacklet, such as the return address, the stack pointer and the arguments pointer (`argp`).

The arguments pointer is needed because the arguments of a function and its activation record may not be contiguous. In sequential execution, the running frame and the parent frame are (usually[2]) contiguous in memory. The caller pushes the arguments of the function it calls onto its activation frame (parent frame) and calls the function, which accesses its arguments through its stack pointer because they lie right above its activation frame. On the contrary, in the stacklet implementation of the cactus stack, the parent frame may not be immediately above the running frame, for example in the case of a stacklet overflow when a new stacklet is allocated. The arguments pointer gives a solution to this problem by keeping a separate pointer to the function's arguments. Just like for the stack pointer, a register is reserved for the arguments pointer.

A worker thread needs to have access to the stacklet stub when checking if a new allocation frame will fit in the stacklet. The frame-pointer register (`$fp`) is employed for that purpose, since it is not used for a language like C. For a language with nested functions, however, the frame pointer would be used, and a different register would have to be reserved to point to the stacklet stub.

Algorithm 5.3 shows how a function prologue is expanded to support stacklets, using sequential function prologue expansion as a subroutine. The sequential expansion involves saving and restoring the callee-saved registers, and updating the stack pointer. The expansion of Algorithm 5.3 is only applied to the parallel clone of a function, leaving the sequential clone optimized for sequential execution. Keep in mind that the stack grows downward, by reducing a global stack pointer (Global_SP) atomically (Line 18). Also, note that in Lines 1-4 the sequential expansion is used

[2]This may not be true for languages that support nested function declarations, but XMTC does not support them.

Algorithm 5.3 Function Prologue Expansion for supporting Stacklets

```
1: if # [Frame_Size = 0 && callee is a leaf function] then
2:     Sequential Function Prologue Expansion
3:     return
4: end if
5: Save Arguments pointer $argp in current activation frame
6: $argp ← $sp                ▷ Keep a pointer to the arguments before updating $sp
7: if #flag [align_cactus_stack] then
8:     EndOfStacklet = $sp && STACKLET_MASK
9: else
10:    EndOfStacklet = $fp.end_address              ▷ $fp points to base of stacklet
11: end if
12: EndOfNewFrame = $sp - frame_size                        ▷ Stack grows downward
13: if EndOfNewFrame < EndOfStacklet then          ▷ Frame doesn't Fit in Stacklet
14:    $fp.saved_sp ← $sp
15:    $fp.saved_argp ← $argp
16:    if $fp.child_stacklet = NULL then                   ▷ Allocate child stacklet
17:        Decrement ← − stacklet_size                       ▷ stacklet_size = 2K
18:        NewStacklet ← psm(Decrement, Global_SP)
19:        $fp.child_stacklet ← NewStacklet
20:        NewStacklet.child_stacklet ← NULL
21:        if #flag [NOT align_cactus_stack] then
22:            NewStacklet.end_address ← NewStacklet-stacklet_size
23:        end if
24:        $fp ← NewStacklet
25:    else
26:        $fp ← $fp.child_stacklet
27:    end if
28:    $sp = $fp - stub_size
29:    $fp.return_address ← $return_register
30:    $return_register ← deallocate_stacklet
31: end if
32: Sequential Function Prologue Expansion + allocate space for saving and restor-
    ing $argp
```

if the function does not need a frame (size=0) and does not call other functions (it is a *leaf function*). The rest of this code checks if the new activation frame needed by the function being called fits in the current stacklet (Line 13), and if not, it allocates a new stacklet off the global stack, using an atomic prefix-sum (fetch-and-add) operation on Line 18. Then, the return address of the function is saved (Line 29)

and set to the stacklet deallocation routine, shown in Algorithm 5.4. The current implementation of the cactus stack does not support freeing and reallocating stacklets within a parallel section, so simply using the prefix-sum is a quick and simple solution for allocating stacklets.

Algorithm 5.4 Stacklet Deallocation

1: `$return_register` ← `$fp`.return_address
2: `$fp` ← `$fp`.parent
3: `$sp` ← `$fp`.saved_sp
4: `$argp` ← `$fp`.saved_argp
5: jump to `$return_register`

The deallocation routine restores the TCU state registers (sp, fp, argp) and returns to the right instruction. In a production implementation, the child stacklet would possibly be returned to a pool of free stacklets. In this prototype, however, this complexity was not necessary because we are keeping stacklets linked and reusing them, as discussed later in this section.

One possible optimization is to align stacklets in memory, and make them all the same size (we picked a size of 2K). That way, checking if a new frame will fit in the current stacklet (Line 13) does not require any memory access: since the size of the frame is a constant, the end of the stacklet can be computed by applying a mask to the stack pointer (`$sp`). The alignment of stacklets is realized by Lines 7-9 and 21-23. Note that these `if` statements control how the expansion is performed and do not appear in the generated code. Empirically, this optimization improves performance modestly (around 2%). One disadvantage of this optimization is that the size of stacklets has to be the same for all functions, including linked libraries. Moreover, if an activation frame is bigger than what a stacklet can fit (in our implementation 2K-stub_size), the compilation fails. In XMTC, this is remedied by recompiling with the optimization that aligns the cactus stack disabled.

In retrospect, we can achieve the same effect of checking if a new activation frame will fit in the current stacklet without accessing memory by placing the stub at

the end of the stacklet instead of the beginning, and having the frame-pointer set at the start of the stacklet stub. In this scenario, the check can be performed by comparing the address of the frame pointer with that of the stack pointer decremented by the size of the requested activation frame.

Another optimization we added on top of Goldstein's original stacklet design is keeping track of the child stacklet owned by the same worker that owns the parent, to avoid the need to free and reallocate it. It is based on the observation that the stack grows and recedes multiple times during a parallel section, especially with recursive codes, which would lead to the frequent allocation, freeing, and reallocation of stacklets. Instead of freeing a child stacklet upon returning, the parent stacklet keeps a pointer to it and reuses it later if needed, as shown by the decision made on Line 16 of Algorithm 5.3. If a worker is unable to allocate a stacklet, unused stacklets can be garbage-collected (and their parent stacklets updated to point to NULL) or even stolen. These more advanced stacklet management techniques are not currently implemented, however. Once a stacklet is allocated in the current XMTC implementation, it will remain under the ownership of the worker that allocated it until the end of the parallel section where it was allocated (i.e., stacklets cannot be returned to a pool of free memory). This has not been a problem in practice because the space allocated to stacklets is reclaimed upon returning to sequential execution, which happens efficiently and frequently on XMT, and because keeping a link to child stacklets allows for good reuse of stacklets.

There are also a type of sequential function clones that need special treatment to support stacklets: functions that introduce parallelism (by means of a spawn statement). Luckily, these are easy to identify because of the *outlining* pass described in Section 5.3.1, which outlines outer spawn statements and places each one in a separate function. Algorithm 5.5 shows the expansion of the prologue of those functions. This expansion does the following: *(1)* it initializes the arguments pointer; *(2)* it allocates an initial stacklet for each TCU; *(3)* if stacklets are aligned, it saves

Algorithm 5.5 Function Prologue Expansion for Outer-Spawn function

```
 1: $argp ← $sp                                          ▷ Set the arguments pointer
 2: Tmp ← $sp − (stacklet_size · #TCUs)                  ▷ Reserve one stacklet per TCU
 3: if #flag [align_cactus_stack] then
 4:     Tmp ← Tmp && STACKLET_MASK
 5: end if
 6: Global_SP ← Tmp
 7: if #flag [align_cactus_stack] then
 8:     $rId ← $sp                  ▷ Save $sp. Use $rId, unused in sequential mode
 9:     $sp ← $sp && STACKLET_MASK  ▷ Necessary because $sp is broadcast
10: end if                          ▷ Right after the join restore the $sp
```

and aligns the stack pointer register of the Master TCU, as it will be broadcast to the parallel TCUs and used as a base for them to access their own initial stacklet; and *(4)* it initializes the global stack pointer to right after the space reserved for the initial TCU stacklets. More specifically, after the arguments pointer is set (Line 1), a stacklet is reserved for each TCU (Line 2) and the global stack pointer (implemented as a regular global variable) is initialized (Line 6), after possibly aligning it (Line 4). Moreover, if the stacklets are aligned, the sequential stack pointer is saved (Line 8) and aligned (Line 9), so that it will be aligned when broadcast to the TCUs. Otherwise, the TCUs would have to perform the alignment, possibly at the beginning of each (outer) task. The stack pointer is restored upon the completion of the spawn statement, as part of the join expansion performed by the post-pass. Lines 1-6 are performed by the core-pass, whereas lines 7-10 are performed by the post-pass as part of the spawn-expansion, similarly to Figure 5.8. Note that the expansion for such outlined functions introducing parallelism should not allocate space on the serial stack. To this end, we had to override GCC's stack allocation and make a special case for such functions.

Finally, some stacklet set-up code needs to be injected at the onset of each parallel section to initialize the stack pointer (`$sp`) and the stacklet pointer (`$fp`) of each TCU. We revisit Figure 5.8 adding the stacklet code in Figure 5.13. Lines 7-9, inserted by the post-pass, initialize the child pointer in the stacklet stub of each

```
1     mvtg   $grHigh , $rHigh
2     getid $rTmp, $rLow
3     mvtg   $grLow , $rTmp
4     spawn
5     broadcast $rLow, $rLow
6     getid $rId , $rLow
7     broadcast $sp , $sp
8     $fp = $sp − (stacklet_size ∗ TCU_ID)
9     $fp.child = NULL
10  spawn_start:
11    chkid $rId , $grHigh
12
13      bcast live regs
14      $fp = $sp − (stacklet_size ∗ TCU_ID)
15      $sp = $fp − stub_size − frame_size
16      BlockAssembly($)
17
18    move $rId , 1
19    ps    $rId , $grLow
20    jump spawn_start
21    join
```

Figure 5.13: Compiling a flat spawn with cactus stack support

TCU. We could have initialized these pointers before switching to parallel mode, but then would have had to perform this sequentially. Note that these child pointers must be initialized before the task code starts on Line 11 because we want to execute it once per TCU per parallel section. Executing it once per task would defeat the purpose of keeping these pointers because it would drop the stacklets allocated to a TCU while it was working on a previous task. Lines 14-15, inserted by the core-pass, show pseudo-assembly that initializes the stacklet pointer (`$fp`) and the stack pointer at the beginning of each task. This could probably be done once per TCU instead of once per task, but the additional overhead is minimal (3 cycles if the stacklet size is a power of 2), and the post-pass, which has access to the per-TCU initialization code (Lines 4-9), does not currently know the size of the activation frame, needed on Line 15. While the post-pass could recover that information, the effort did not seem worthwhile.

So far we have discussed the implementation of the cactus stack in the absence of nested parallelism. Later in this chapter, we will revisit it to include support for

nested parallelism.

5.6.3 Function Insertion

As mentioned in Section 5.3.3, in the current prototype of XMT, *all* the code that may be executed by the TCUs must be laid out between the `spawn` and the `join` instructions, including the code of functions that may be called. A pass called *function insertion* builds a call graph at link-time (in the post-pass), and for each outer spawn block, inserts before the join instruction the code of the functions that may be called. This is different to function inlining because the code of the function is simply placed within the spawn-join that may call it, and not substituted at every call-site.

This pass is simple enough, but because a function may be called from different spawn statements within a code, the inserted clones of the functions must be updated to have unique assembly labels. Remember that function pointers are not supported in XMTC, so complex pointer analysis is not needed to build the call-graph. However, as we will see, nested spawns get compiled to code that uses function pointers in a limited way. To accommodate that, we include in the call-graph functions whose address is taken, without performing any further pointer analysis.

5.6.4 Dead Function Elimination

So far, we have created two clones for each function. We then inserted function clones in the assembly between the spawn and join instructions for each outer spawn statement. This creation of many clones for each function blows up the size of the code. To remedy that, we eliminate all the function clones that cannot possibly be called by an application, after the function insertion pass. Among others, this includes all parallel clones of functions because only the inserted clones (see Section 5.6.3) are actually called. This helps to reduce significantly the size of the

produced binary and `.sim` files, and to reduce the compilation time, as the post-pass is the slowest component, even though it performs the simplest analyses.

5.6.5 Outlining Optimizations

The outlining pass isolates outer spawn statements so that the core-pass (GCC) does not perform illegal optimizations, as discussed in Section 5.3.1. Outlining introduces inefficiencies by creating a function call where there was none. One way to take advantage of this additional function call is to pass global variables as arguments as well, in a controlled way. At first, this sounds counter intuitive: why pass as arguments variables that are global and therefore directly accessible by the tasks? The reason is that, according to the XMT calling convention (inherited from MIPS), the first four arguments to a function are passed in registers. We can take advantage of this convention by passing as arguments global variables that are read-only in the task code, as long as we do not exceed four arguments. These global values will be passed through registers and then broadcast to the tasks, hence avoiding the creation of a hot spot in memory.

Currently, this optimization is not path-sensitive, as it will not differentiate between read-only variables accessed by the task code to try and select the ones that may be accessed more frequently (e.g., ones accessed in loops and ones not accessed in conditionally executed code). This would be beneficial when the candidate global variables exceed the four arguments passed through registers. A different optimization would be to explicitly load these values in registers in the outlined function, so that they can be broadcast. Such an optimization is harder to implement, however, as it needs both high level information about the code and low level register allocation information available in the back-end of GCC. An easier alternative would be to load such variables in the read-only cache at the cluster level. The global variable would then be accessed once by each cluster, which is not as good as just once by the Master TCU, but does not increase register pressure.

Another potential optimization has to do with reducing the number of arguments the outlined function needs, in order to make space for passing and broadcasting global values, as per the previous optimization. Sometimes, the low and high expressions determining the number of tasks of a spawn statement can be complex, involving multiple variables, as shown in Figure 5.14(a). In that case, hoisting those expressions, as shown in Figure 5.14(b), can reduce the number of arguments significantly. Of course, if performed naively, hoisting can increase the number of arguments needed by the outlined function by two. A simple solution is to run outlining once with hoisting and once without, and to select the option that requires fewer arguments. This is not currently implemented, and therefore, the hoisting pass is not enabled by default.

(a) original code	**(b) after hoisting**
	`low = a+b*c/d;`
	`high = e+f*g/h;`
`spawn (a+b*c/d, e+f*g/h){ // 8 args`	`spawn (low, high){ // 2 args`
`    task_code($)`	`    task_code($)`
`}`	`}`

Figure 5.14: Hoisting Low & High Expressions

Other ways of reducing the overhead of outlining include *(1)* better analysis for reducing the number of arguments (e.g., finding the smallest number of variables or subexpressions to describe all the incoming and outgoing values of the outlined function), and *(2)* inlining back these functions at link-time. However, inlining at that late stage, after register allocation has been performed, can be challenging, and we have not explored that possibility.

5.6.6 Compiling a nested spawn statement

So far, we have described support for calling functions from parallel code and the necessary cactus stack support. In this section, we will add the necessary

components to support nested spawn statements. Algorithm 5.6 shows pseudocode of a parallel clone of a function with a parallel loop (e.g., spawn statement). To allow potentially executing some of the tasks of that parallel loop on other workers, we perform a different type of outlining for nested spawns. This time, we only include the code of the task, excluding the spawn statement, since the goal is to be able to run one or more tasks independently. Algorithm 5.7 shows the resulting pseudocode which includes two functions: the original function `FOO` that has been modified in several ways, and the outlined function that executes the task code *grain* times.

Algorithm 5.6 Outlining Inner Spawn: original code

1: **procedure** FOO($params_{foo}$)
2: $locals_{foo}$
3: SomeCode
4: **for all** $\$ \in [low, high]$ **do**
5: CODE[$params_{foo}, locals_{foo}, \$$]
6: **end for**
7: RestOfCode
8: **end procedure**

Let us focus first on the outlined function. The dollar sign represents the (register-allocated) task identifier. The scheduler sets its value before invoking the outlined function. The *grain* argument allows executing multiple tasks with one invocation, a very useful feature for fine-grained codes. The $frame$ argument gives access to FOO's local variables and parameters. The task code (Line 20) is updated to access those variables through the frame pointer. The $frame$ is defined separately for each function containing a `spawn`, as a structure (i.e., a C `struct`) that contains all the local variables of FOO, its parameters, and some additional fields for bookkeeping, which we will cover later.

The code of FOO is also updated to access its variables through the frame structure (Lines 4 and 15). The overhead of doing so is smaller than what one would originally assume because FOO will access the fields of the $frame$ structure directly and not through a pointer. This means that the compiler can access any

Algorithm 5.7 Outlining Inner Spawn: resulting code

```
1: procedure FOO(params_foo)
2:     frame = (state_regs, rop, nChildren, tid, params_foo, locals_foo)
3:     frame ← params_foo
4:     SomeCode[frame]
5:     frame.nChildren ← high − low + 1                ▷ Start of Inserted Code
6:     frame.tid ← $
7:     frame.state_regs ← (sp, fp, argp, rop)
8:     taskDescriptor ← (low, nChildren, frame, outlinedNestedSpawn, grain)
9:     nExec ← schedulerExecute(taskDescriptor)
10:    atomic{frame.nChildren ← frame.nChildren − nExec}
11:    if nChildren ≠ 0 then
12:        Jump to Scheduler Code
13:    end if
14:    ClobberRegisters                                 ▷ End of Inserted Code
15:    RestOfCode[frame]
16: end procedure
17: procedure OUTLINEDNESTEDSPAWN(grain, frame)
18:    end ← $ + grain   ▷ $ ← td.low by the run-time before calling this function
19:    while $ < end do
20:        CODE[frame.params_foo, frame.locals_foo, $]
21:        $ ← $ + 1
22:    end while
23: end procedure
```

of the fields of $frame$ using its stack pointer, as $frame$ is now a local variable in FOO. Moreover, the compiler can register-allocate fields of $frame$ like it would the variables of the original instance of FOO. Hence, the main overhead of this transformation is copying FOO's parameters into the $frame$ structure (Line 3). Copying is necessary because, as we mentioned, the arguments of a function may not lie directly above its activation frame, due to the cactus stack implementation.

Besides local variables and arguments, the frame structures of all functions that have a parallel loop also have a common preamble (Figure 5.15). This preamble is used by the scheduler for synchronization and for allowing the worker executing the last pending task of the parallel loop to resume the parent task. In that case, the preamble is needed only if the worker resuming the parent task is not the one that initiated the spawn statement. The scheduler code can type-cast any frame

```
typedef struct {
    void (*rop)(void);   // Rest Of Parent pointer
    int sp;              // stack pointer
    int fp;              // frame pointer
    int argp;            // args pointer
    int childNR;         // number of pending tasks
    int tid;             // task ID of parent task
} genericFrame;
```

Figure 5.15: Common Preamble of all frame structures.

to a `genericFrame` in order to access the fields of the preamble because all frames have the same preamble. Note that the first argument is a function pointer that takes no arguments. It is used to jump to the next instruction after the parallel loop (i.e., spawn statement), which is Line 14 in Algorithm 5.7. Line 14 clobbers all but the state registers (`$sp`, `$fp`, `$argp`, `$`) so that GCC will restore any register-allocated values from the stack. This seems unnecessary as the address of the $frame$ is passed to the scheduler on Line 9, and therefore, any register-allocated fields of $frame$ need to be written out to memory before that call and restored afterwards. However, GCC may decide to restore these registers soon after returning from the scheduler code, before Line 14. In that case, a remote worker jumping to that line will start executing code which incorrectly assumes that some of the fields of $frame$ reside in registers. Clobbering the registers at that point ensures that GCC will not try to restore fields of $frame$ into registers too soon.

```
typedef struct {
    int tId;          // TID of first thread
    int tNr;          // Number of tasks
    int grain;        // Grainsize
    void *frame;      // Pointer to parent frame
    /** The address of the code to execute.
     * The 1st argument is the grainsize.
     * The 2nd argument is the pointer to the frame
     * of the parent to access its variables. */
    int (*code_addr)(int grain, void* frame);
} TaskDescriptor;
```

Figure 5.16: The task descriptor structure

Now, let us focus on the code that replaced the nested spawn statement (Lines

5-13 of Algorithm 5.7). First, the number of child tasks is initialized in the $frame$ (Line 5), and the task ID of the current task is saved (Line 6). Then, the rest of the frame preamble is filled (Line 7). Remember that the *rop* is the address of Line 14. A task descriptor structure is then initialized on Line 8. A task descriptor is akin to a *closure*: it contains all the information needed to execute some code. Here, the code is a set of tasks. Figure 5.16 shows the fields of the task descriptor of a parallel loop. The ID of the first task to execute along with the total number of tasks define the range of tasks included in a task descriptor. The pointer to the $frame$ allows the task code to access its parent's local variables, and a function pointer to the task code allows to execute tasks. Lastly, the $grain$, which is computed by the static coarsening pass or is explicitly provided by the programmer, allows the task code to execute multiple tasks in a single go.

After this initialization of the parent's frame preamble and of the task descriptor, the scheduler is called on Line 9 (see Algorithm 4.2) to schedule and execute the task descriptor. The scheduler returns the number of tasks that were executed by the present worker ($nExec$). If some tasks were placed on the deque, $nExec$ will be smaller than $nChildren$, the number of tasks originally in the task descriptor. Since other workers may be working on a subset of those tasks after stealing them, they will have access to $frame.nChildren$, which they will also decrement, as they complete tasks. For that reason, the operation on Line 10 is performed atomically. In Section 5.6.7, we will present the code that decrements $nChildren$ in the case of stolen tasks executing remotely, as well as how the thief resumes the parent task if needed (Algorithm 5.8). Finally, the parent task checks if all its tasks have completed (Line 11) and resumes if that is the case (Line 14) or jumps to scheduler code responsible for feeding the worker with more work. In work stealing, the main family of schedulers implemented in XMTC, that is the code for stealing code from a victim worker. Note that the worker does not *call* the scheduler code, it simply jumps to it, relinquishing its stack to the worker that completes the last of the

child-tasks and becomes responsible for resuming the parent task.

5.6.7 Outer Spawn Compilation for Nesting

Now, we want to combine XMT's hardware scheduling of outer spawn statements with software scheduling for nested parallelism. The first step is to convert the spawn statement, as done in Figure 5.17. Depending on the task ID ($), the worker will either execute it, or turn to the software scheduler for work if the task ID is not a valid outer task ($\$ > high$ on Line 2). In the current implementation, incrementing $high$ within the spawn block is not supported because our goal is to support structured nested parallelism. Therefore, the legacy single-spawn (`sspawn`) is not supported. Moreover, since $high$ cannot be increased, TCUs will keep requesting work from the scheduler once outer tasks are consumed, hence the infinite loop on Line 4.

(a) Original Code

```
spawn (low, high) {
    CODE[$]
}
```

(b) Converted Code

```
1  spawn (low, high) {
2      if ($>high) {
3          Allocate Stacklet
4          while(true) {
5              Get Work from scheduler
6              Execute it
7              If(last) resume parent
8          }
9      } else {
10         CODE[$]
11     }
12 }
```

Figure 5.17: Outer Spawn Conversion

Before we elaborate on the stacklet allocation (Line 3) and on the body of the scheduling loop (Lines 5-7), we address the issue of detecting global termination of a parallel section in the presence of nested parallelism. We use the same XMT hardware support as we did for a flat spawn statement in Figure 5.8, but we use it differently, as shown in Figure 5.18. We use three global registers in-

```
1    mvtg   $grConsumed, 1
2    mvtg   $grProduced, 0
3    getid $rTmp, $rLow
4    mvtg   $grLow, $rTmp
5    spawn
6    broadcast $rLow, $rLow
7    getid $rId, $rLow
8    broadcast $sp, $sp
9    $fp = $sp - (stacklet_size * TCU_ID)
10   $fp.child = NULL
11  spawn_start:
12   bcast live regs
13   $fp = $sp - (stacklet_size * TCU_ID)
14   $sp = $fp - stub_size - frame_size
15   if ($ > high) {
16   } else {
17     BlockAssembly($)
18   }
19  join_label:
20   call
21   move $rId, 1
22   ps   $rId, $grLow
23   jump spawn_start
24   join
```

Figure 5.18: Compiling an outer spawn to support nesting

stead of two. The use of `$grLow` is the same as before; it stores the next available task ID for outer tasks. However, we do not use `$grHigh` in a `chkid` instruction to detect the global termination of the parallel section, as nested tasks may be pending even if the last outer task has completed. Instead, we use two global registers: `$grProduced` (Line 2), which is increased each time a task descriptor is added onto the scheduler's work-pool, and `$grConsumed` (Line 1), which is increased each time a task descriptor is removed from the scheduler's work-pool. The difference of `$grProduced-$grConsumed` gives the number of available task descriptors in the work-pool. Thanks to XMT's hardware prefix-sum support, this information can be maintained at very low cost and in a scalable way. We use `chkid $grConsumed, $grProcuced` to block workers when the work-pool is empty. In reality we first copy the value of `$grConsumed` into a local register using a prefix-sum with a zero increment, because `chkid` takes a local register and a fixed global register as arguments. The reason for using two global registers is that the hardware prefix-sum

operation does not accept a decrement at the moment. If it did, we could simply keep the total number of task descriptors available in the work-pool. Such support would be needed in a production quality XMT to avoid overflow, but for the current prototyping stage, it has not been a limitation in practice. Note that `$grConsumed` is initialized to 1 instead of 0 because `chkid` blocks only when its first argument is strictly greater than its second argument. When they are equal, execution proceeds with the next instruction.

Using XMT's `chkid` instruction allows to block TCUs when there is no work in the work-pool. In work stealing, where the work-pool is distributed and workers have to randomly probe the deques of other workers, the `chkid` mechanism prevents wasted probing, which can potentially be harmful for performance by issuing useless memory requests. Moreover, the `chkid` mechanism effectively notifies idle workers when a task descriptor is added so that they can resume work stealing. Work stealing implementations on platforms that do not have such hardware support employ methods such as exponential back-off after a certain number of unsuccessful deque probes to avoid adversely affecting the performance of busy workers.

Algorithm 5.8 Scheduling loop for work stealing

1: **while** true **do**
2: $success \leftarrow stealWork(\&taskDescriptor)$
3: **if** $success = false$ **then** continue with next iteration
4: **end if**
5: $nExec \leftarrow schedulerExecute(taskDescriptor)$
6: $gFrame \leftarrow (_xmt_generic_frame)\ taskDescriptor.frame$
7: $atomic\{gFrame.nChildren \leftarrow gFrame.nChildren - nExec\}$
8: **if** $gFrame.nChildren = 0$ **then** ▷ Resume Parent Task
9: `$sp` $\leftarrow gFrame.sp$ ▷ Restore Stack Pointer
10: `$fp` $\leftarrow gFrame.fp$ ▷ Restore Frame Pointer
11: `$argp` $\leftarrow gFrame.argp$ ▷ Restore Arguments Pointer
12: `$` $\leftarrow gFrame.tId$ ▷ Restore Task ID
13: jump to $gFrame.rop$ ▷ Jump to Rest of Parent Task
14: **else**
15: Keep executing work of own deque
16: **end if**
17: **end while**

Algorithm 5.8 expands Lines 5-7 of Figure 5.17(b) for work stealing schedulers. These include Cilk's scheduler, TBB's simple-partitioner, auto-partitioner and affinity-partitioner (not implemented in XMTC), and the lazy work stealing variants described in the present dissertation. First, the worker attempts to steal a task descriptor until it succeeds (Lines 2-3). Then, the scheduler executes the descriptor, which involves the possibility of splitting it and pushing part of it on the local deque (Line 5). Upon return, the worker atomically decrements the number of children of the parent task through the frame pointer stored in the descriptor (Lines 6-7), and if the worker just completed the last task of the parent task, it resumes it (Lines 8-13). Otherwise, the worker consumes the local deque (Line 15) that may have been filled by the call on Line 5 if the task descriptor was split.

Chapter 6

Model of Scheduling Costs and Architectural Support

As a quick reminder, this dissertation focuses on approaches that improve the efficiency of declarative structured parallel code, i.e., code that expresses all available parallelism using structured constructs that can be nested, such as parallel loops, sum-like reductions, and parallel function calls or futures. Declarative code typically exposes much more parallelism (in the form of short tasks) than there are hardware resources, and it falls upon the compiler and run-time to map this abundance of parallelism efficiently onto longer executable threads as best fits the machine at hand. In Chapter 3, we identified two distinct goals of coarsening, amortizing scheduling costs, and pruning parallelism, the first of which falls mainly onto the compiler since, by the time parallelism reaches the scheduler, it should be coarse enough so that scheduling overheads are amortized.

To make it possible for the compiler to amortize scheduling costs, it must be able to do two things: *(1)* estimate the cost of each task with relative accuracy, and *(2)* know approximately the cost of parallelizing and scheduling a set of tasks. This chapter focuses on the latter aspect. We present a simple *parametric* model for scheduling overheads and propose a methodology for validating this model. The idea is to have an auto-tuner figure out the parameters of the model by using micro-benchmarks when the compiler is installed on a platform, but this is beyond the scope of this dissertation. Instead, we manually run benchmarks to determine the values of the parameters on the XMT platform. We also give some initial evidence that the model is expressive and accurate enough to answer questions such as "is it preferable to execute a parallel loop sequentially, or to keep it parallel?".

The proposed model covers two schedulers, the popular work stealing software scheduler and XMT's hardware scheduler, as well as their combination. A

contribution of this work is to show the *orthogonality* of the two schedulers: given a certain amount of parallelism in the code, transitioning from XMT's hardware scheduling to software scheduling mid-course still allows software to exploit the full advantage of the parallelism not capitalized by the hardware. We also show that XMT's added hardware scheduling benefits performance, both in the model and experimentally. Without this performance improvement, observing the orthogonality of hardware and software scheduling on XMT is without consequence. With it, however, lightweight hardware support for scheduling becomes beneficial and synergistic with existing software approaches.

The hardware scheduling support on XMT allows it to simultaneously initialize all its hardware threads and to assign them to pending tasks as they become available, at very low cost. This enables XMT to even take advantage of small amounts of parallelism. For example, Breadth First Search (BFS), shown in Figure 6.1, can be implemented as a series of rounds, each uncovering the next layer of a graph, starting from an initial vertex. The parallelism in each round depends on the number of vertices in the layer being processed (`thisLevel`) and the number of edges incident on them. Therefore, depending on the graph characteristics, each round may contain small amounts of parallelism, even for large graphs, which could preclude the possibility of efficient parallelization. On XMT, however, the hardware scheduler allows harnessing even such low-degree parallelism to produce good speedups that appear not to be possible on other platforms (e.g., [19]).

On the other hand, XMT's hardware scheduler does not directly support nested parallelism. Instead, a hybrid approach that relays nested parallelism to a software scheduler is implemented (see Chapter 5). Nevertheless, the software scheduler takes advantage of some of the features of the hardware scheduler, such as global termination detection and a mechanism for putting cores (TCUs) to sleep when there is no work available for them to execute. In other words, the hybrid approach combines the best of the hardware and software schedulers. In contrast, as a

means for avoiding spin-waiting, existing scheduling implementations on multicores either update some global state, an approach that is not very scalable, or have idle workers (that try to steal but fail to find any work) yield control to the operating system.

```
void bfs(Graph g(v,e), Vertex root) {
  VertexSet thisLevel={root}, nextLevel={};
  int level = 0;
  while (thisLevel!={}) {
    parfor (Vertex v in thisLevel) {
      parfor (Vertex u in v.neighbors) {
        if (u.level == NOT_SET &&
           u.getekeeper.try_acquire()==true) {
          nextLevel.atomicAdd(u);
          u.level = level;
          u.gatekeeper.release(); // optional
    } } }
    level += 1;
    thisLevel=nextLevel; nextLevel={};
} }
```

Figure 6.1: A common case for low-degree parallelism: BFS

In this chapter, we make a first step towards formalizing a model for task scheduling. We propose a model for a hardware and a software scheduler, use it to predict performance for a toy example and validate this prediction experimentally. Another contribution of this work is that we observe an *orthogonality* relationship between the two schedulers on XMT, both in the model and experimentally. Specifically, given any amount of parallelism in code, a transition from hardware to software scheduling at any time during the execution still allows the software scheduler to get the most from the parallelism left unexploited by hardware. This provides a strategy for combining the strength of the hardware and software schedulers in one

hybrid system. This study indicates that adding hardware is important for supporting declarative parallel code efficiently. Furthermore, this work *(1)* sets a direction for developing a predictive model of scheduling costs, and *(2)* presents preliminary evidence that building such a model is feasible.

Bringing this line of work to maturity would involve (1) extending the model to cover more types of schedulers, (2) validating it on multiple platforms using several benchmarks, (3) implementing an auto-tuner to automatically discover the values of the model parameters, and (4) implementing the model internally in a compiler infrastructure and making it accessible to compiler passes whose role will be to decide how to execute the asset of available parallelism optimally. In short, completing this work could very well be an additional dissertation on its own. This chapter presents work in preparation for a workshop submission [84].

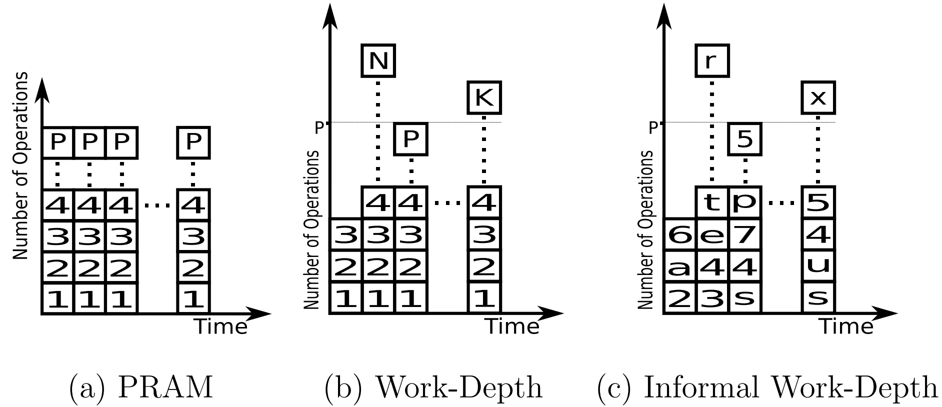
6.1 Background

6.1.1 Algorithmic Models

Since ease-of-programming is our first order concern, we present some background on the algorithmic model that will be translated into a declarative parallel program. We start our presentation from the PRAM model and elevate to the more abstract *Work-Depth* and *Informal Work-Depth* models, so as to reach the level of abstraction we desire declarative parallel programmers to enjoy.

PRAM [50, 55] is the most developed parallel algorithmic model. It assumes P *synchronous* processors and constant-time memory access. A PRAM algorithm is a sequence of time steps, each performing exactly P operations (Figure 6.2(a)). Hardware synchrony is unrealistic for today's multi-threaded machines, but research on parallel programming education suggests that relaxing the PRAM synchrony to write declarative task-parallel code is straightforward [82]. It reports that teaching parallel algorithmic thinking based on PRAM algorithms and asking students to

write task-parallel code sparked their creativity and allowed them to tackle more complex problems. The catalyst in this case was the good performance XMT achieved on declarative parallel code.



(a) PRAM (b) Work-Depth (c) Informal Work-Depth

Figure 6.2: Increasingly abstract algorithmic models

The academic community has expressed concerns that PRAM is too simplistic to be implemented, especially with regards to the unit time memory access. The scope of these concerns has to be carefully delimited. Given today's technology, these concerns are true for distributed computing, where uniform constant-time memory access is unrealistic, or for multi-chip parallel computing, where inter-chip communication bandwidth is limited and the latency is high. For these platforms, more complex models (e.g., LogP [30], Bulk Synchronous PRAM [85]) are arguably necessary to achieve significant efficiency but also harder for the algorithm designer and programmer. On the other hand, the ongoing growth of on-chip silicon real-estate has recently allowed to build complex multi-cores and many-cores on a single chip, and while not all memory locations are equidistant even on single-chip parallel processors, modeling them as such can be a reasonable and useful first approximation because the difference between the closest and the furthest (on-chip) memory location is negligible compared to that of distributed machines. Our claim is not that the assumption of unit time memory access is realistic for existing commercial multi-cores and many-cores, but merely that a single-chip parallel platform can be built to cater to that assumption. XMT supports this claim by taking advantage of

this abundance of on-chip resources to efficiently support parallel code derived from PRAM algorithms [82, 73]. Therefore, we hold that adequate architectural support can make PRAM a realistic abstraction, and not one that is too simplistic.

In fact, we claim that PRAM is not too simple, but too hard! It is undesirable for the programmer or algorithm designer to take P, the number of processors, into account. Instead, in the Work-Depth model (WD) [78], a program is a sequence of time steps, each performing *any* number of operations. Figure 6.2(b) illustrates the difference between WD and PRAM: a time step can have fewer or more operations than the number of processors P. The WD model maps directly to PRAM without hiding additional complexities, for which it requires the operations of each time step to be sequentially numbered. This requirement conflicts with our desire that high-level parallel code should allow nested parallelism. Imagine the scenario where two tasks T_1 and T_2 are active in time-step-1, creating n and m additional tasks. In time-step-2, we will have $n + m + 2$ tasks, but the algorithm designer/programmer has to oversee their sequential numbering. For example, T_1 sends n to T_2; then T_1 numbers its new tasks $T_3, \ldots, T_{n+2}$ and T_2 numbers hers $T_{n+3}, \ldots, T_{n+m+2}$.

To overcome this limitation, the Informal Work-Depth model (IWD), recently popularized as the Immediate Concurrent Execution (ICE) abstraction [86], allows the operations of each time-step to be described as (unordered) sets (see Figure 6.2(c)). Among other things, this allows nesting of parallelism without accounting for its scheduling costs. The IWD model provides the level of abstraction and flexibility we want programmers and algorithm designers to enjoy, but for it to be realistic, the compiler, run-time and hardware platform must be able to translate such declarative parallel code to efficient execution. If necessary, the programmer is certainly allowed to explicitly order the operations within a set like they would have to do for the WD model (e.g., for parallel I/O).

The IWD model (like WD) characterizes algorithms via two metrics, *work* and *depth*. *Depth* D (also known as time, span, or length of the critical path) is the total

number of time-steps of the algorithm, and the lower-bound on its execution time given an infinite number of processors. The Depth can also be thought of as the longest path through the computation DAG (Directed Acyclic Graph). *Work* W is the total number of operations performed over the whole computation. In the work stealing bounds, we used the symbols T_∞ for depth and T_1 for work. Using these two metrics, $\frac{W}{D}$ is a measure of the parallelism expressed by the algorithm. If $\frac{W}{D} >> P$, we say that the algorithm has sufficient parallelism for a system with P processors. This is the parallel slackness assumption discussed in Section 4.10.

What is still missing is proper theoretical underpinning of the following question: assuming that the implementation platform is a variable, what is the realm of possibilities for implementing IWD parallelism? Specifically for nested parallelism, what are the costs of software and hardware primitives the run-time system can employ and how can the compiler transform the code to maximize performance? In addition to trying to lay out such a theoretical framework, the proposed model unravels the relationship between the hardware and software-based schedulers on XMT. The orthogonality relationship we establish between them enables harnessing both for a better combined solution.

6.1.2 Background for Modeled Schedulers

In this chapter, we focus on two specific schedulers, as it is beyond the scope of this work to cover a broader spectrum of schedulers. The two schedulers are XMT's hardware scheduler and the popular software work stealing scheduler, which has been implemented for commercial platforms, as well as for XMT. In this section, we briefly reiterate on some XMT and work stealing background needed for the rest of the chapter. The expert or sequential reader of this dissertation is invited to skip the rest of this section.

To understand XMT's hardware scheduler, recall that XMT is an *asymmetric* many-core: it has one powerful master core and *many* lightweight cores. The

powerful core is used for sequential code or for the sequential portions of a parallel code, while the lightweight cores are used for parallel portions. Any execution starts on the master core (*sequential mode*), and when a parallel section is encountered, its execution is delegated to the parallel cores (*parallel mode*). When the parallel section is completed, execution resumes on the master core (*sequential mode*).

To efficiently support quick transitions between sequential and parallel mode, XMT provides hardware scheduling which includes: *(1)* broadcasting the instructions of the parallel section by the master core to the parallel cores; *(2)* a hardware primitive called *prefix-sum* to allow multiple parallel cores to each grab the next available tasks simultaneously with constant overhead; *(3)* a hardware mechanism that detects the termination of a parallel section and initiates the transition to sequential mode (akin to a barrier operation, but over tasks instead of parallel cores). The prefix-sum primitive allows fine-grained scheduling because of its very low cost. Also, note that the latency of the prefix-sum operation is logarithmic in the number of parallel cores P, but it is a constant for a given platform because P is constant for a given platform and because concurrent prefix-sum operations are not queued but serviced concurrently.

XMT's hardware scheduler has one limitation: it does not directly support nested parallelism. If additional nested parallelism is created while in parallel mode, the hardware scheduler cannot directly schedule it efficiently, as it implements a global queue abstraction (FIFO), which is known to have a potentially unbounded memory footprint. Therefore, the hardware only schedules *outer parallelism*, and the XMT system provides software task-management for *nested parallelism*.

The lazy work stealing scheduler [83] described in Chapter 4 is one of the available schedulers on XMT, but for simplicity we consider the non-adaptive, traditional work stealing (e.g., Cilk [38]). *Work Stealing* has gained popularity in academia and industry for its good performance, ease of implementation and good theoretical space and time bounds. The basic idea of work stealing is for each worker thread to

place tasks, when discovered, in a local *deque* (a work-pool data structure), greedily perform that work from its local deque, and *stealing* work from the deques of remote processors when the local deque is empty. When a vector of N tasks is discovered (e.g., through a parallel loop such as in Listing 6.1), the processor iteratively splits the vector and pushes half on the deque. At the end of this process, the deque contains $\log N$ task vectors. The premise is that, as these vectors get stolen and the same splitting process is performed on the thief threads, the distribution of parallelism resembles a binary tree of depth $\log N$ with N single-task leaves.

6.2 A Work-Depth Model for XMT's Hardware and Work Stealing Software Schedulers

To propose a model for the schedulers described in Section 6.1.2 we start by considering the simple but powerful parallel-loop construct shown in Listing 6.1. Without loss of generality, we will assume that `low=0`, `step=1`, and `high=N`. Generally `N` is *not* known at compile-time and is often input-dependent. According to the IWD model, the *depth* of the parallel loop in our example is the maximum among the depths of its N tasks, $D = \max_i Depth(Code(i))$, and the *work* is the sum of the works of its N tasks, $W = \sum_i Work(Code(i))$. Next, these equations are incremented with the scheduling costs.

Listing 6.1: Generic Parallel Loop

```
for all(int i=low; i<high; i+=step){ Code(i); }
```

For the hardware scheduler there is a cost for switching to parallel mode and initializing all parallel cores, as well as a synchronization cost for switching back to serial mode at the end. We model the sum of these costs as a constant (Q). There is also a cost associated with the fine-grained assignment of cores to tasks. We model it as a constant cost (a) per-task. The cost Q is incurred once for each parallel

loop, and it is on the critical path, so it is added both to the depth and the work in Table 6.1. The cost a is incurred once per task, so it appears N times in the work, and only once in the depth, because the hardware scheduler does not incur extra overhead such as queuing to assign cores to tasks *concurrently*.

	Depth	**Work**
IWD	$D = \max_i Depth(CODE(i))$	$W = \sum_i Work(Code(i))$
Hardware	$D_{HW} = D + Q + a$	$W_{HW} = W + Q + aN$
Software	$D_{SW} = D + Q + b \log N$	$W_{SW} = W + Q + bN$

Table 6.1: WD equations for a parallel-loop with N tasks

For work stealing, the same cost Q of switching between modes is incurred on XMT. For an x86 multi-core, a different cost Q' is incurred for initializing the parallel threads and for synchronizing at the end of a parallel section, but measuring that cost is beyond our scope. There is also a management cost b per task, which we model as a constant. Like a, it appears N times in the work, but in the depth it appears with a $\log N$ factor, which emanates from the iterative splitting of the task-vector, described in Section 6.1.2. Note that the model for work stealing is more or less implicit in may studies discussing the asymptotic behavior of work stealing schedulers. The contribution of this work is to propose a way to determine the constant factors involved.

At this point, we have a cost model for parallel loops using XMT's hardware scheduler and the more general work stealing scheduler. Note that the quantities Q, a, and b are platform and implementation dependent. For example, on an x86 multi-core, b may have different values for TBB [76] and TPL [61], both work stealing schedulers. Many questions arise, such as how to model other hardware and software schedulers, or how to model other parallel constructs like futures and reducers. We defer those to future work and focus on the following question: *is this limited model*

useful? For example, assuming that we know the values of Q, a, and b on our target platform XMT, and the values for work and depth (W and D) of a parallel loop, can we use the model to decide whether to use hardware, software, or a hybrid scheduling approach, or even to run the code sequentially? In the next section, we address this question using matrix multiplication as an example parallel code.

6.3 Evaluation of the Scheduling Models

To demonstrate the potential of our model, we must be able to measure the quantities Q, a, and b and they must have reasonably low variability to justify modeling them as constants. We use the familiar computation of matrix multiplication as a running example for measuring these constants. Figure 6.3 shows a possible parallel implementation. It takes two matrices $A_{N\times L}$ and $B_{L\times M}$ as inputs and stores the result in matrix $R_{N\times M}$. In this example, the two outer loops are parallelized: for each element of the result matrix R the computation is performed in parallel. The innermost loop can also be parallelized using a *reducer* [37], but for simplicity we do not do this.

According to IWD, the *work* is proportional to the number of multiplications: $W = wNML$, where w is the cost per multiplication. The *depth* is the amount of work executed by each parallel task: $D = wL$. When only the loop over i is parallelized, the depth is $D_i = wML$.

Table 6.2 presents the work and depth equations for our matrix multiplication example. Since XMT's hardware scheduler can only take advantage of outer parallelism, the depth is D_i, which is M times larger than D. To get the best of both hardware and software schedulers, we consider a hybrid scheduler that uses hardware for outer parallelism and software scheduling for nested parallelism.

For the work equations, all approaches pay the constant cost Q, then a cost per task scheduled, a for the hardware and b for the software. The software scheduler pays a cost of bN for the outer parallel loop and a cost of bNM for the inner one.

```
void matmult(int A[N][L], int B[L][M],int R[N][M]){
  for all (int i=0; i<N; i++) {
    for all(int j=0; j<M; j++) {
      R[i][j] = 0;
      for (int k=0; k<L; k++) {
        R[i][j] += A[i][k]*B[k][j];
} } } }
```

Figure 6.3: Parallel Matrix Multiplication

Work	$W = wNML$
Hardware	$W_{HW} = W + Q + aN$
Software	$W_{SW} = W + Q + bN + bNM$
Hybrid	$W_{Hyb} = W + Q + aN + bNM$

Depth	$D = wL, \quad D_i = wML = MD$	**Parallelism**
Hardware	$D_{HW} = MD + Q + a$	N
Software	$D_{SW} = D + Q + b \log N + b \log M$	NM
Hybrid	$D_{Hyb} = D + Q + a + b \log M$	NM

Table 6.2: Work and depth equations for hardware, software, and hybrid scheduling.

The cost Q is also found in all the depth equations, and the logarithmic overheads for software scheduling stem from the iterative splitting described in Section 6.1.2.

6.3.1 Measuring Q, a, b

Since the parameters Q, a, and b are platform and implementation specific, the values computed will only be valid on the chosen experimental platform, in this case the XMT FPGA [92]. Table 6.3 presents the values of Q, a and b computed experimentally using square matrices ($N = M = L$) that fit in the shared cache.

N	8	16	32	64
Q	251.0	251.0	251.0	251.0
a	23.3	24.7	23.7	25.7
b	1515.5	1256.3	1162.4	1189.8

Table 6.3: Values in cycles for different values of N (with $N = M = L$).

We find that Q is constant, and that the values of a and b are relatively independent of N, which justifies modeling them as constants. The methodology followed to get these measurements as well as a discussion on how to measure the constants a and b more accurately are presented in the next section.

We envision automatically determining the parameters Q, a and b by using a series of predetermined benchmarks when the compiler is installed. The values will then be used by compiler and run-time optimizations that transform the code to better map the fine-grained tasks to the (long running) worker threads. For example, our static analysis estimates the number of cycles for fine-grained tasks (i.e., their depth), and it combines them to make tasks of at least 1000 cycles. According to our findings that the average cost of software scheduling per task is $\tilde{b} = 1281$, and assuming the cycle estimation is accurate, it appears beneficial to perform more aggressive coarsening.

6.3.2 Methodology for measuring Q, a, b

Table 6.4 presents the computed values for Q, a, b, and w along with the measured values that were used to compute them.

Measuring Q. To measure Q, we replaced the body of the computation by a parallel loop with zero iterations and measured its execution. This caused execution to switch to parallel mode and immediately back to serial. This was repeated with different input sizes confirming that the initialization of matrices `A` and `B` preceding

N	8	16	32	64
Q	251	251	251	251
W_{HW}	27767	210855	1659143	13184967
$W_{Ser/Par}$	27604	210484	1658408	13183348
a	23.3	24.7	23.7	25.7
w	53.4	51.3	50.6	50.3
W_{SW}	39705	230560	1695580	13259468
D_{SW}	12271	22945	63361	227425
b	1515.5	1256.3	1162.4	1189.8
b_{steal}	2867.9	2389.0	2258.7	3531.4

Table 6.4: Values are in cycles for different values of N (with $N = M = L$).

the main computation does not affect the cost Q. This indicates that the cost Q does not depend on the state of the machine before the transition.

Measuring a. To compute the parameter a of XMT's hardware scheduler, we measured the parallel work W_{HW} and the sequential work W and subtracted the measured values. However, since XMT is an asymmetric architecture and parallel cores are different than the sequential one, the factor w might have a different value on the sequential and parallel cores. Therefore, we ran the *sequential code* on a parallel core and measured $W_{Ser/Par} = Q + a + W$. We computed a by solving $W_{HW} - W_{Ser/Par} = (N - 1) \cdot a$.

To compute the work factor w, we solved $W_{Ser/Par} - Q - a = wNML$ for w and used the values for Q and a previously computed.

Measuring b. To compute the parameter b, we measured the work and depth of the computation using the software scheduler. To simplify the process, we serialized the `j`-loop and only kept the `i`-loop parallel. The work and depth equations became $W_{SW} = Q + bN + wNML$ and $D_{SW} = Q + b \log N + wML$. To measure the work,

we measured the execution time of the parallel code on a single parallel core. To measure the depth, we measured the execution time using all parallel cores, but we ensured that the number of tasks did not exceed the number of parallel cores. We solved W_{SW} for b using the computed values for Q and w and also solve D_{SW} for b and call it b_{steal}. We notice an important divergence in the b values computed stemming from the different behavior of work stealing in these two cases. In the first case (W_{SW}), only one worker is active, so no thefts occur: the worker pushes and pops tasks from its own work-queue. In the second case (D_{SW}), multiple workers are active and tasks are always stolen, which is a more costly operation. Furthermore, in the case where $N = 64$, the last worker to steal a task is going to be looking for a single task among 63 work-queues. This is the worst possible scenario for work-stealing, as there is typically an abundance of work, so it can be found quickly (parallel slackness assumption). Both values, b and b_{steal}, are useful because the first reveals the common-case cost of scheduling per task, whereas the second gives an upper bound on the worst case.

There are, however, some caveats to this method. In retrospect, using synthetic benchmarks to measure the values of Q, a, and b (and b_{steal}) would have been preferable. This way we would exclude queuing overheads incurred by the multiplication operations, or having many write operations in flight in the interconnection network delaying the switch back to sequential mode, or the inaccuracy of using $W = wNML$ instead of the more accurate $W = wNML + sNM$ to account for the NM store operations updating the result matrix. In fact, if we assign $s = 30$ cycles, which is approximately the cost of a round-trip through the interconnection network on the FPGA, computing w gives 49.63, 49.44, 49.66, and 49.82, for sizes 8, 16, 32, and 54, respectively. These values for w have a much lower variation than the ones we computed earlier. However, picking $s = 30$ cycles long seems high since store operations are non-blocking, but this long latency (of 30 cycles) could be explained by the hardware buffers filling up with non-blocking store requests, resulting in most

of the store operations blocking. In conclusion, synthetic benchmarks would have avoided these issues. Even so, the values of a and b show relatively little variation, so we decided not to repeat the experiments with synthetic benchmarks.

6.3.3 Orthogonality of Hardware and Software Scheduling on XMT

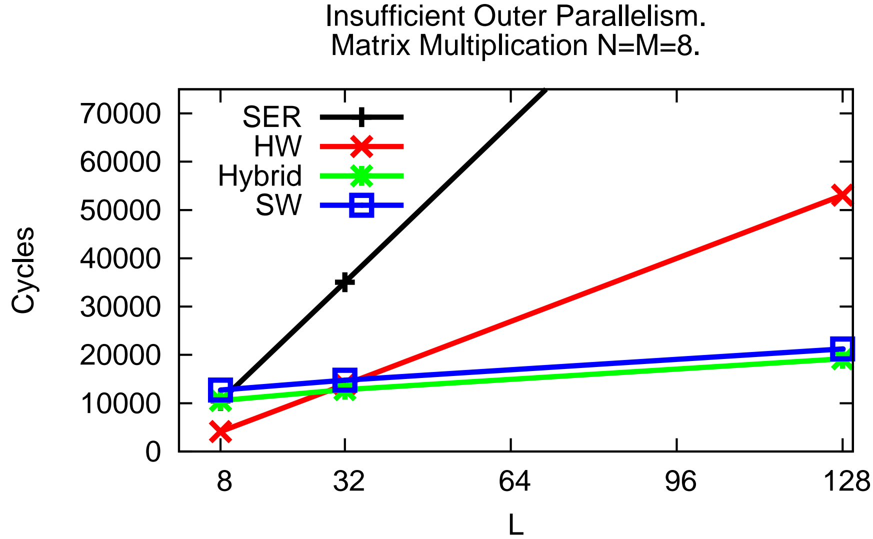


Figure 6.4: Hybrid vs. hardware: insufficient outer parallelism.

The formulas in Table 6.2, supported by experimental data in Figures 6.4, 6.5, and 6.6, provide the basis for characterizing of the relationship between hardware and software scheduling on XMT as *orthogonal*. Given any amount of flat parallelism (not nested) in code, transitioning from hardware to software scheduling at any time during the execution still allows the software to fully exploit the parallelism left unexploited by the hardware. In other words, the combination of hardware and software scheduling (i.e., hybrid scheduling) should never be slower than the latter. In fact, Table 6.2 demonstrates this for the more challenging case where the parallelism delegated to the software scheduler is not flat: given that $a < b$, hybrid scheduling is always preferable to software scheduling since $W_{Hyb} < W_{SW}$ and $D_{Hyb} < D_{SW}$. This is also confirmed experimentally by comparing the two

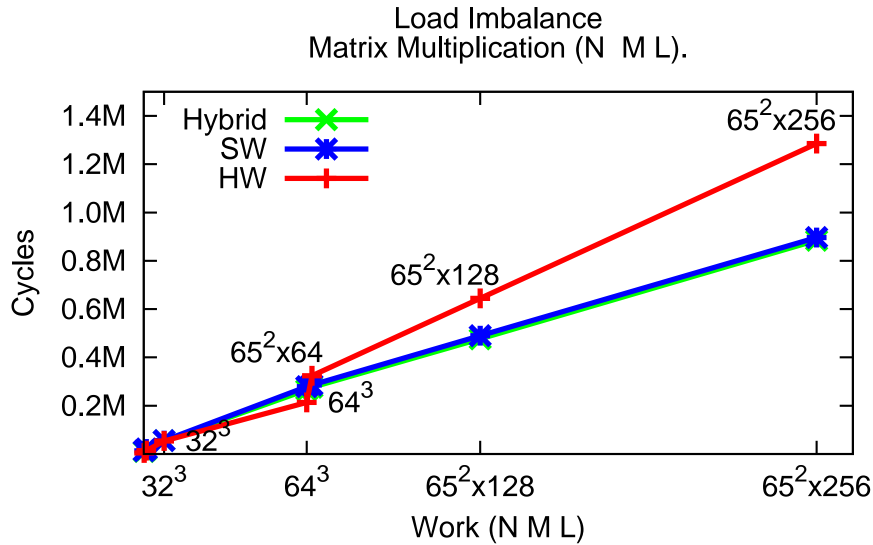


Figure 6.5: Hybrid outperforms hardware: load imbalance.

approaches in Figures 6.4, 6.5, and 6.6.

This orthogonality relationship shows the *composability* of XMT's hardware scheduler with work stealing to create a hybrid scheduler that is always faster than its software component alone. Hereafter, we present quantitative examples where the hybrid solution outperforms the exclusive use of hardware scheduling, but also examples where hardware scheduling is faster. The existence of both examples further advocates the utility of adding architectural support for scheduling and the need for a theoretical model for predicting the costs of hardware and hybrid scheduling to assist the compiler or run-time in selecting between the two options.

Hybrid scheduling incurs a larger work overhead than hardware scheduling (by bNM in our example), which is the additional cost of nested parallelism. The potential advantage of more parallelism is expressed by the smaller depth $D = D_i/M$ for the hybrid scheduler. Thus, hardware scheduling incurs less work-overhead than hybrid scheduling, but it can have a much larger depth, as it does not exploit nested parallelism. When the depth is the deciding factor for performance, for example when there is not enough outer parallelism to feed the available workers (Figure 6.4)

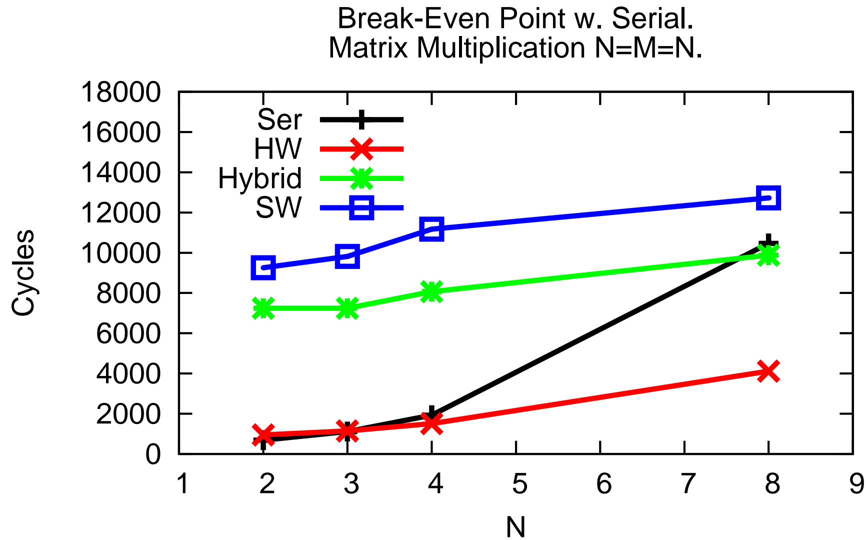


Figure 6.6: Break-even point with sequential code.

or to ensure good load balance (Figure 6.5), hybrid scheduling is preferable to hardware. On the other hand, when work is the deciding factor (e.g., abundant and balanced outer parallelism or very short parallel sections, as in Figure 6.6) hardware scheduling comes ahead.

In particular, Figure 6.4 shows an example where hybrid scheduling can outperform hardware because outer parallelism ($N = 8$) is insufficient to employ all 64 threads, but along with inner parallelism ($M = 8$) it creates 64 tasks, enough to feed the 64 parallel cores of our experimental platform. By varying L, we find that hybrid scheduling outperforms hardware scheduling, for values over 25.

Figure 6.5 illustrates another example where hybrid scheduling comes ahead. We plot the running times of seven (N, M, L) configurations as a function of their (normalized) work $W/w = NML$. The seven configurations are (8^3), (16^3), (32^3), (64^3), $(65^2, 64)$, $(65^2, 128)$, $(65^2, 256)$. The key point to notice is the step-increase for hardware scheduling when moving from 64 to 65 outer tasks. This occurs because the hardware schedules the first 64 tasks in parallel, then, the first worker to complete its task runs the 65th task, creating an imbalance since the other workers will finish and

idle. Consequently, hybrid scheduling, which takes advantage of nested parallelism and creates more tasks, is a winner for these configurations by achieving better load balance.

On the other hand, Figure 6.6 shows that the hardware scheduler gets ahead of the serial execution for $N = M = L = 4$ ($W/w = 4^3$), whereas hybrid scheduling needs at least $N = M = L = 8$ ($W/w = 8^3$, i.e., eight times more work). This shows the advantage of having dedicated hardware for exploiting low-degree or fine-grained parallelism. Such hardware can enable parallelizing certain kinds of irregular computations, such as breadth first search (BFS), especially on graphs that do not have the small-world property (i.e., low diameter), such as planar graphs. In the next section, we use this model to predict which thread management technique will be preferable in the scenarios of Figures 6.4 and 6.6.

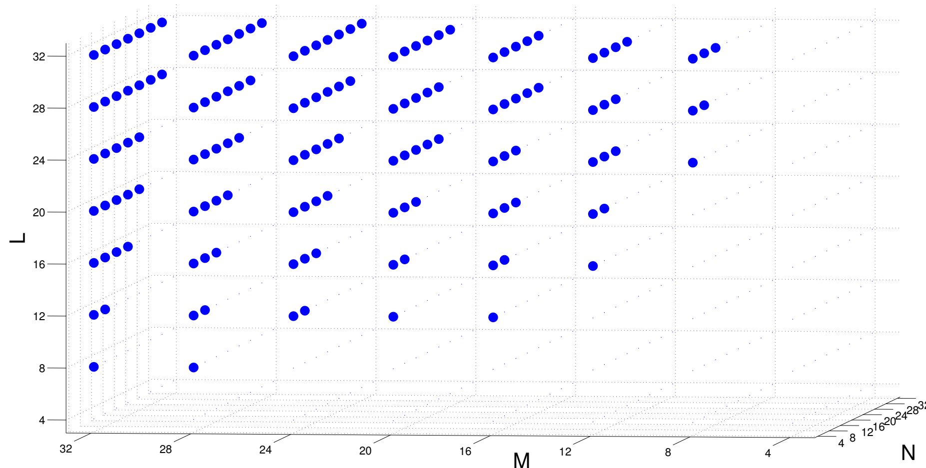


Figure 6.7: Hardware and hybrid scheduling are complementary.

Finally, Figure 6.7 shows a 3D scatter plot of configurations for which hybrid scheduling is faster than hardware scheduling. The axes represent values of N, M, and $L \in \{4, 8, 12, 16, 20, 24, 28, 32\}$. The blue dots represent configurations for which hybrid scheduling is faster than hardware, and the absence of a dot means that hardware scheduling outperforms hybrid for that input size. Hybrid scheduling wins

when there is not enough outer parallelism (N is small) and the inner parallelism is worth exploiting, i.e., when the product $ML \sim D_i$ is large enough. On the other hand, hybrid scheduling is slower than hardware scheduling when there is enough outer parallelism (N is large enough), or when there is not enough inner parallelism (M is small and hybrid scheduling can not take advantage of much more parallelism) and the depth is small (L is small and it is not as profitable to take advantage of the nested parallelism because the tasks are too short), i.e., when the product $ML \sim D_i$ is small.

6.3.4 Using the model to predict the preferable scheduling option

We will use the following values: $Q = 250$, $a = 25$, $b = 3000$ (which are slightly higher than the average values measured, as we prefer to slightly overestimate rather than underestimate the scheduling overheads), and $w = 50$.

First, in the case illustrated in Figure 6.4 (N=M=8), the runtime for the hardware scheduler will be:

$$T_{HW}(L) = D_{HW}(L) = wML + Q + a = 400L + 275$$

For hybrid scheduling, the runtime will be:

$$T_{Hyb}(L) = D_{Hyb}(L) = wL + Q + a + b \log M = 50L + 9275$$

Solving $T_{HW}(L) - T_{Hyb}(L) = 0$, gives $L = 25.7$ so the model predicts that for $L \geq 26$ hybrid scheduling is preferable, which is very close to the cutoff point of 25 determined experimentally.

Second, in the case illustrated in Figure 6.6, we address when hardware and hybrid scheduling will outperform the sequential execution. We have $N = M = L$. For $N \leq 8$ (i.e., $N \cdot M \leq nTCUs$), the following equations give the running times for the three approaches:

$$T_{ser}(N) = 50N^3$$

$$T_{HW}(N) = 50N^2 + 275$$

$$T_{Hyb}(N) = 50N + 3000\lceil \log N \rceil + 275$$

$T_{ser}(N) - T_{HW}(N) = 0$ gives $N = 2.16$, so for $N \geq 3$ hardware scheduling outperforms sequential execution, which is accurate. Solving $T_{ser}(N) - T_{Hyb}(N) = 0$ gives $N = 6.211$, which means that for $N \geq 7$ hybrid scheduling outperforms serial execution, which is very close to the actual value of $N = 8$.

6.4 Related Work

He et al. present the Cilkview scalability analyzer [45], a very useful tool for developers of parallel code. Cilkview dynamically instruments optimized binaries: when the binary is run normally, no overhead is payed, but when it is run in "profiling-mode" it is instrumented on-the-fly. Cilkview runs an instrumented program on one processor and computes the work, the span (i.e., depth) and the *burdened span* of the application, which charges the cost of a theft for each possible task. Using these metrics, Cilkview plots the optimal and the worst-case expected performance as a function of the number of workers. It also automatically benchmarks the application and includes the data collected on the aforementioned plot. The goal is to assist the programmer in determining why a parallel application might not perform well. Cilkview will find if the application lacks parallelism, or if the execution times are below Cilkview's worst-case estimated performance, the bottleneck is likely somewhere else (e.g., memory bandwidth). Cilkview is the only tool we are aware of that tries to model the scheduling overheads (burdened span). However, its goals do not include the accurate modeling of scheduling overheads in order to allow a compiler to make coarsening decisions. Instead, it is meant to be an interactive debugging tool. He et al. [45] also review other debugging tools for multi-threaded codes, including mostly profilers, which do not attempt to model the scheduler overheads.

Li et al. [63] propose to add hardware support for scheduling, but their approach does not seem to be composable with a software approach. More importantly, the scheduling algorithm has the potential to deviate from depth-first execution, which can drastically increase the memory footprint. While they raise some interesting ideas, overall the approach seems questionable.

Kumar et al. [57] propose to add hardware support for work stealing, but they employ a centralized unit to hold all the deques. This is somewhat counter-intuitive, because work stealing is a distributed algorithm by design. Furthermore, the authors admit that their proposed design is not scalable and they do not mention whether they support nested parallelism or not. Like the hardware scheduler proposed by Li et al. [63], this one also does not seem to be composable with a software approach.

6.5 Conclusions and Future Directions

In this chapter, we identified the abstraction level at which we want programmers to operate, and we made three contributions towards efficiently supporting it. First, we presented a case for architectural support of scheduling by showing cases where hardware scheduling is superior to software or even hybrid scheduling. Second, we showed that the hardware and software schedulers we considered have orthogonal contributions to performance. Third, we introduced a parametric model for two scheduling approaches and their combination (hybrid scheduling) and experimentally evaluated their parameters on XMT using a simple example. This led to the identification of some typical cases where hybrid scheduling outperforms hardware and vice versa. The long-term goal is to build a generalized model of scheduling overheads that will guide the compiler and run-time system in doing transformations to optimize declarative code for each specific target platform. For example, the model would allow the compiler to (statically or dynamically) decide among serializing parallelism, coarsening parallelism (possibly by serializing nested parallelism), flattening nested parallelism, or interchanging nested parallel-loops to

expose more parallelism to the hardware scheduler.

Chapter 7

Conclusion and Future Directions

This dissertation focused on techniques for efficiently supporting declarative code as a means to supporting general-purpose parallel programming. An integral part of declarative code is the ability to express nested parallelism, which creates the need for dynamic scheduling. Work stealing is currently the scheduler of choice for such task-parallel codes, because it allows programmers to expose much more parallelism than there are cores on the target platform. However, just as with any other dynamic scheduler, when tasks are too fine-grained, the scheduling overheads kill the performance. To avoid that situation, programmers are currently required to manually coarsen the available parallelism and only expose some of it.

We identified two goals that coarsening needs to fulfill: amortizing scheduling overheads by coarsening very fine-grained tasks, and pruning excess parallelism which is unlikely to help improve performance. We showed that current manual approaches are tedious and either result in loss of performance portability, or require expert programmers who will spend a lot of time tuning their coarsening approach to preserve performance portability. We proposed an experimental framework for measuring performance portability and used it to demonstrate the pitfalls of manual coarsening. We also presented some simple but effective compiler transformations that amortize very fine-grained parallel loops.

To tackle pruning of parallelism, the more complex of the two goals of coarsening, we presented *lazy scheduling*, an adaptive scheduling technique that takes load conditions into account. We showed that it is competitive with existing work stealing schedulers on coarse code and greatly outperforms them on declarative and on amortized codes, where only the very fine-grained tasks were coarsened either manually or by compiler. We presented results on three commercial multicores as

well as the XMT platform, and showed the importance of honoring the *breadth-first thefts* order of work stealing, especially on commercial multicores.

We also presented details of the XMTC compiler, which enabled doing research on the XMT platform and has been used to teach PRAM algorithms and XMT to students from high-school to graduate school. We also presented a number of lessons learned while modifying a compiler for a sequential language (GCC) to compile an explicitly parallel language such as XMTC.

Finally, we proposed a model for scheduling costs for the XMT hardware scheduler. We also presented a model for work stealing that has been implicit in some previous works. The main contributions of that portion of our work are: (1) a validation of the model and a methodology for computing its constant parameters; (2) the composability of XMT's hardware scheduler and with work stealing that makes their hybrid composition better than the sum of its parts.

We hope that the overarching contribution of this dissertation was to restore the shaken belief that declarative programming can be efficiently supported and to show how the current approach of manual coarsening, besides being tedious, can easily destroy the performance portability of parallel code. With these points in mind, we hope that this work will serve as a basis for future research towards efficiently supporting declarative programming.

7.1 Future Work

We see a few possible directions towards better support of declarative programming. First, our compiler cannot currently amortize all types of codes. In particular, recursively nested parallelism eludes it. Thus, one of the pieces currently missing from the puzzle is a set of compiler transformations that will amortize all types of task parallel code, starting with recursively nested parallelism.

Once this support is in place, the next step would be to extend the work we did on modeling the scheduling costs to enable the compiler to decide between different

ways to exploit the exposed parallelism. In our envisioned solution, an auto-tuner would determine the parameter values of the model, as well as any parameter values needed for the cost estimation pass described in Section 3.5.1.

Another area where future work will be needed is in identifying the exact conditions under which lazy work stealing has the same bounds as work stealing. This will dispel any doubts about the behavior of lazy work stealing under worst-case scenarios that may have eluded its experimental evaluation in this dissertation.

Finally, supporting declarative programming is but one of the aspects of solving general purpose parallel programming. We believe that other aspects will include: (1) adding hardware support for parallelism, such as scheduling on XMT; (2) automatically increasing the determinism of parallel code to simplify debugging; (3) reducing the degree to which the programmer is responsible for programming for locality, as is the case with XMT and with another recently described architecture [27]; (4) coordinating between the OS and the application for resource allocation and re-allocation during execution, as in [5, 74]; (5) improving dynamic memory allocation and garbage collection for declarative parallel applications. This is not meant to be a complete list of challenges for general purpose parallelism, but rather, a short list of significant items demonstrating the magnitude of such an enterprise.

Bibliography

[1] Intel Threading Building Blocks Reference Manual, Rev. 1.9, 2008.

[2] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, SPAA '00, pages 1–12, New York, NY, USA, 2000. ACM.

[3] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Oracle scheduling: controlling granularity in implicitly parallel languages. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 499–518, New York, NY, USA, 2011. ACM.

[4] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[5] Kunal Agrawal, Charles E. Leiserson, Yuxiong He, and Wen Jing Hsu. Adaptive work-stealing with parallelism feedback. *ACM Trans. Comput. Syst.*, 26:7:1–7:32, September 2008.

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

[7] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, Berkeley, Dec 2006.

[8] D H Bailey, E Barszcz, J T Barton, R L Carter, T A Lasinski, D S Browning, L Dagum, R A Fatoohi, P O Frederickson, and R S Schreiber. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[9] A. O. Balkan, Gang Qu, and U. Vishkin. An area-efficient high-throughput hybrid interconnection network for single-chip parallel processing. In *Proc. Design Automation Conference*, pages 435–440, June 2008.

[10] A. O. Balkan, Gang Qu, and Uzi Vishkin. A mesh-of-trees interconnection network for single-chip parallel processing. In *Proc. IEEE Intl. Conf. on Application-specific Systems, Architectures and Processors*, 2006.

[11] Aydin O. Balkan. *Mesh-of-trees Interconnection Network for an Explicitly Multi-threaded Parallel Computer Architecture*. PhD thesis, University of Maryland, 2008.

[12] Aydin O. Balkan, Michael N. Horak, Gang Qu, and Uzi Vishkin. Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. In *Proc. of Hot Interconnects*, 2007.

[13] Lars Bergstrom, Mike Rainey, John Reppy, Adam Shaw, and Matthew Fluet. Lazy tree splitting. In *Proc. of the 15th International Conference on Functional Programming - ICFP '10*, Sept., 2010.

[14] Guy Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[15] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 102–111, New York, NY, USA, 1993. ACM.

[16] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

[17] Hans-J. Boehm. Threads cannot be implemented as a library. In *Proc. Conference on Programming Language Design and Implementation*, 2005.

[18] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, FPCA '81, pages 187–194, New York, NY, USA, 1981. ACM.

[19] George Caragea, Fuat Keceli, Alexandros Tzannes, and Uzi Vishkin. General-purpose vs. GPU: Comparison of many-cores on irregular workloads. In *Proc. USENIX Workshop on Hot Topics in Parallelsim*, 2010.

[20] George Caragea, Alexandre Tzannes, Fuat Keceli, Rajeev Barua, and Uzi Vishkin. Resource-aware compiler prefetching for many-cores. In *Proc. Intl. Symposium on Parallel and Distributed Computing*, 2010.

[21] George C. Caragea. *Optimizing for a Many-Core Architecture without Compromising Ease-of-Programming.* PhD thesis, University of Maryland, College Park, 2011.

[22] George C. Caragea, Beliz Saybasili, Xingzhi Wen, and Uzi Vishkin. Performance potential of an easy-to-program pram-on-chip prototype versus state-of-the-art processor. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures*, 2009.

[23] George C. Caragea, Alexandros Tzannes, Aydin O. Balkan, and Uzi Vishkin. XMT Toolchain Manual for XMTC Language, XMTC Compiler, XMT Simulator and Paraleap XMT FPGA Computer, 2010.

[24] George Constantin Caragea and Uzi Vishkin. Brief announcement: better speedups for parallel max-flow. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 131–134, New York, NY, USA, 2011. ACM.

[25] Olivier Certner, Zheng Li, Pierre Palatin, Olivier Temam, Frederic Arzel, and Nathalie Drach. A practical approach for reconciling high and predictable performance in non-regular parallel programs. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 740–745, New York, NY, USA, 2008. ACM.

[26] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[27] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In Lawrence Rauchwerger and Vivek Sarkar, editors, *PACT*, pages 155–166. IEEE Computer Society, 2011.

[28] R Cole and U Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, STOC '86, pages 206–219, New York, NY, USA, 1986. ACM.

[29] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[30] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[31] D.E. Culler, J.P. Singh, and A. Gupta. *Parallel computer architecture: a hardware/software approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, 1999.

[32] Edsger W. Dijkstra. Go-to statement considered harmful. *Commun. ACM*, 11:147–148, 1968.

[33] T. M. DuBois, B. Lee, Yi Wang, M. Olano, and U. Vishkin. XMT-GPU: A PRAM architecture for graphics computation. In *Proc. International Conference on Parallel Processing*, 2008.

[34] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An adaptive cut-off for task parallelism. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[35] Alejandro Duran, Marc Gonzàlez, and Julita Corbalán. Automatic thread distribution for nested parallelism in openmp. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 121–130, New York, NY, USA, 2005. ACM.

[36] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup versus efficiency in parallel systems. *Computers, IEEE Transactions on*, 38(3):408–423, mar 1989.

[37] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures*, 2009.

[38] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the conference on Programming Language Design and Implementation (PLDI)*, 1998.

[39] Etienne M. Gagnon and Laurie J. Hendren. Sablecc, an object-oriented compiler framework. In *Proc. Technology of Object-Oriented Languages*, 1998.

[40] Seth Copen Goldstein. *Lazy Threads Compiler and Runtime Structures for Fine-Grained Parallel Programming.* PhD thesis, University of California–Berkeley, Berkeley, CA, 1997.

[41] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996.

[42] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The nyu ultracomputer–designing a mimd, shared-memory parallel machine (extended abstract). In *Proceedings of the 9th annual symposium on Computer Architecture*, ISCA '82, pages 27–42, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[43] Yi Guo, Jisheng Zhao, V. Cave, and V. Sarkar. Slaw: A scalable locality-aware adaptive work-stealing scheduler. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[44] Robert H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 9–17, New York, NY, USA, 1984. ACM.

[45] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. The cilkview scalability analyzer. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 145–156, New York, NY, USA, 2010. ACM.

[46] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized non-blocking work stealing deque. *Distrib. Comput.*, 18:189–207, February 2006.

[47] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 280–289, New York, NY, USA, 2002. ACM.

[48] Lorin Hochstein, Victor R. Basili, Uzi Vishkin, and John Gilbert. A pilot study to compare programming effort for two parallel programming models. *J. Syst. Softw.*, 81:1920–1930, November 2008.

[49] Sunpgack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration for multi-core cpu and gpu. In *Proc. of The Twentieth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[50] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[51] F. Keceli, T. Moreshet, and U. Vishkin. Thermal management of a many-core processor under fine-grained parallelism. In *Proc. 5th Workshop on Highly Parallel processing on Chip(HPPC 2011)*, Bordeaux, France, August 2011. In conjunction with Euro-Par.

[52] Fuat Keceli. *Power and Performance Studies of the Explicit Multi-Threading (XMT) Architecture*. PhD thesis, University of Maryland, College Park, 2011.

[53] Fuat Keceli, Tali Moreshet, and Uzi Vishkin. Power-performance comparison of single-task driven many-cores. In *17th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2011.

[54] Fuat Keceli, Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Toolchain for Programming, Simulating and Studying the XMT Many-Core Architecture. In *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (in conjunction with IEEE IPDPS)*, Anchorage, Alaska, USA, May 2011.

[55] Jorg Keller, Christopher Kessler, and Jesper Larsson Traeff. *Practical PRAM Programming*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[56] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: a high-performance parallel lisp. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, pages 81–90, New York, NY, USA, 1989. ACM.

[57] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 162–173, New York, NY, USA, 2007. ACM.

[58] Leslie Lamport. Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[59] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[60] D. Leijen and J. Hall. Optimize Managed Code For Multi-Core Machines. *MSDN Magazine*, October 2007.

[61] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM.

[62] Charles E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM.

[63] Zheng Li, Olivier Certner, Jose Duato, and Olivier Temam. Scalable hardware support for conditional parallelization. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 157–168, New York, NY, USA, 2010. ACM.

[64] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proc. ACM Symposium on Principles of Programming Languages*, 2005.

[65] Xavier Martorell, Eduard Ayguadé, Nacho Navarro, Julita Corbalán, Marc González, and Jesús Labarta. Thread fork/join techniques for multi-level parallelism exploitation in numa multiprocessors. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 294–301, New York, NY, USA, 1999. ACM.

[66] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 45–54, New York, NY, USA, 2009. ACM.

[67] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 185–197, New York, NY, USA, 1990. ACM.

[68] Dorit Naishlos, Joseph Nuzman, Chau-Wen Tseng, and Uzi Vishkin. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '01, pages 93–102, New York, NY, USA, 2001. ACM.

[69] Dorit Naishlos, Joseph Nuzman, Chau-Wen Tseng, and Uzi Vishkin. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. *Theory of Computing Systems*, 36:521–552, 2003. 10.1007/s00224-003-1086-6.

[70] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, 2002. Springer-Verlag.

[71] Dimitrios S. Nikolopoulos, Eleftherios D. Polychronopoulos, and Theodore S. Papatheodorou. Efficient runtime thread management for the nano-threads programming model. In *Proc. of the Second IEEE IPPS/SPDP Workshop on Runtime Systems for Parallel Programming, LNCS*, pages 183–194, 1998.

[72] OpenMP Architecture Review Board. OpenMP Application Program Interface, Ver. 3.0 May 2008. `http://www.openmp.org`.

[73] D. Padua, U. Vishkin, and J. C. Carver. Joint UIUC/UMD parallel algorithms/programming course. In *Proc. NSF/TCPP Workshop on Parallel and Distributed Computing Education, in conjunction with IPDPS*, 2011.

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

[75] David Patterson. The trouble with multicore: Chipmakers are busy designing microprocessors that most programmers can't handle. *IEEE Spectrum*, July 2010.

[76] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in tbb. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –8, april 2008.

[77] A. Beliz Saybasili, Alexandros Tzannes, Bernard R. Brooks, and Uzi Vishkin. Highly parallel multi-dimentional fast fourier transform on fine- and coarse-grained many-core approaches. In *PDCS '09: The 21st IASTED International Conference on Parallel and Distributed Computing and Systems*, 2009.

[78] Yossi Shiloach and Uzi Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *J. Algorithms*, 3:128–146, February 1982.

[79] Marc Snir. Multi-core and parallel programming: Is the sky falling?, 2008. http://www.cccblog.org/2008/11/17/multi-core-and-parallel-programming-is-the-sky-falling/.

[80] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

[81] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. Stackthreads/mp: integrating futures into calling standards. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '99, pages 60–71, New York, NY, USA, 1999. ACM.

[82] Shane Torbert, Uzi Vishkin, Ron Tzur, and David J. Ellison. Is teaching parallel algorithmic thinking to high school students possible?: one teacher's experience. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 290–294, New York, NY, USA, 2010. ACM.

[83] Alexandros Tzannes, G.C. Caragea, R. Barua, and U. Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP*. ACM, 2010.

[84] Alexandros Tzannes, Uzi Vishkin, and Rajeev Barua. A case for architectural support for task management. manuscript, April 2011.

[85] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[86] Uzi Vishkin. Using simple abstraction to reinvent computing for parallelism. *Commun. ACM*, 54:75–85, January 2011.

[87] Uzi Vishkin, George C. Caragea, and Bryant C. Lee. *Handbook of Parallel Computing: Models, Algorithms and Applications*, chapter Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform. CRC Press, 2007.

[88] Uzi Vishkin, Shlomit Dascal, Efraim Berkovich, and Joseph Nuzman. Explicit multi-threading (xmt) bridging models for instruction parallelism (extended abstract). In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 140–151, New York, NY, USA, 1998. ACM.

[89] J. Wagner, A. Jahanpanah, and J.L. Traff. User-land work stealing schedulers: Towards a standard. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, pages 811 –816, march 2008.

[90] Xingzhi Wen. *Hardware Design, Prototyping and Studies of the Explicit Multi-Threading (XMT) Paradigm*. PhD thesis, University of Maryland, College Park, 2008.

[91] Xingzhi Wen and Uzi Vishkin. PRAM-on-chip: first commitment to silicon. In *Proceedings of the annual ACM Symposium on Parallel Algorithms and Architectures*, 2007.

[92] Xingzhi Wen and Uzi Vishkin. FPGA-based prototype of a PRAM-on-chip processor. In *Proceedings of the 2008 conference on Computing frontiers - CF '08*, page 55, May 2008.