PROGRAMMING MODELS FOR SPECULATIVE AND OPTIMISTIC PARALLELISM BASED ON ALGORITHMIC PROPERTIES

A Thesis
Presented to
The Academic Faculty

by

Romain Cledat

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

Georgia Institute of Technology
December 2011

PROGRAMMING MODELS FOR SPECULATIVE AND OPTIMISTIC PARALLELISM BASED ON ALGORITHMIC PROPERTIES

Approved by:

Professor Santosh Pande,
Committee Chair
School of Computer Science
Georgia Institute of Technology

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

Professor Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Professor Umakishore Ramachandran
School of Computer Science
Georgia Institute of Technology

Professor Karsten Schwan
School of Computer Science
Georgia Institute of Technology

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

Date Approved: 22 August 2011

In memoriam my father, Bertrand Cledat,

For my family, in particular for my Mom who has given me so much,

For my new companion in life, Vishakha

ACKNOWLEDGEMENTS

I first arrived at Georgia Tech in August of 2004, for a Masters in ECE. Coming from France, I was completely lost: the size of the Georgia Tech campus was daunting, the number of administrative steps required to simply sign up for classes overwhelming... Today, it is amusing to see myself in the lost students that show up mid-August. Seven years separate me from those new students trying to find their way. Looking back, I realize how much I have changed, hopefully in a positive way. I owe this change, in large part, to the people who supported me over the years. Naming all of you would be impossible but you know who you are; just know that I would not have made it through some of the most trying times of my life without you.

Before I get into the detailed acknowledgment of those people, I must express my most heartfelt thanks to my family. As a kid, I had the immense opportunity to live in another country: having lived my first eight years in France, my Dad, due to his experience as a Masters student in the US, asked for his transfer to the US where we stayed for five years. Upon my return to France, I knew that I wanted to return. I thank my parents for giving me this taste for the US as well as for piquing my curiosity about other cultures and other ways of life. I would also like to thank my parents in particular, and my brothers, for supporting me throughout my education: I was always afforded the best opportunities and allowed to pursue my dreams. This PhD is particularly dedicated to my Dad. Although he will not see me walk, I realize today how similar I am to him. I owe a lot of my inquisitiveness, my pursuit for perfection and my voracious curiosity for pretty much anything to him. Thank you Dad.

Several people contributed to my success during my years at Georgia Tech. I owe

particular thanks to my wife Vishakha Gupta. Even though I met her only in 2006, she has celebrated with me my successes and most importantly, has stood by me and helped me overcome my doubts, frustrations, sadness and failures. Through all these ups and downs, she has been a constant beacon of stability and temperance. Her constant care and love have kept me afloat on more than one occasion.

I also owe particular thank to many of my initial friends and colleagues who "initiated" me to the life of a PhD student (and yes, PhD Comics is uncannily accurate). Many thanks to Lakshmi Chakrapani, who, like Mike Slackenerny, first introduced me to all the delicacies of PhD life. He was aided by Yogesh Chobe, Jaswanth Sreeram, Rick Copeland and Tushar Kumar who also contributed in gently easing me into the PhD program. Thank you for your friendship and for providing laughs and useful advice. I would also like to thank Pinar Korkmaz who was part of our research group.

I would also like to thank my current friends and colleagues, Jaswanth Sreeram, Kaushik Ravichandran, Sangho Lee, Changhee Jung and in particular Tushar Kumar. While Tushar provided me with invaluable insights and help on my work, I am most grateful for his advice and support on a variety of topics which helped me see things more clearly on multiple occasions.

Over the years, I have also been blessed with a supportive circle of friends: Celine Lascar, Vincent Combes, Gerrit Becker, Pei Yoong Koh, Adit Ranadive, Smita Vaidya, Mukil Kesavan, Raghav Vijaywargiya, Rakshita Agrawal, Dulloor Rao, Ruchi Anand, Nawaf Almoosa, Karishma Babu, Jui Deshpande, Ashwin Kolhe, Gregory Diamos, Sudnya Padalikar, Amit Tambe, Danesh Irani, Bhuvan Bamba, Vivek Sharma, Sanjay Kumar, Madhumitha Ravichandran, everyone previously cited and many others. I would also like to particularly thank my roommates and "pseudo-roommates": Gerrit Becker, Yogesh Chobe, Matthew Konopa, Jaswanth Sreeram, Vijay Balasubramanian, Madhavi Wagh, Sucharita Otta and Priyanka Tembey. Thank you for bearing with all my idiosyncrasies and sharing your life with me.

I would like to express my heartfelt gratitude to my advisor, Professor Santosh Pande. His guidance and prodding made this thesis what it is today. His advice made me the research I am today, feeling prepared to tackle new and complex problems and contribute in a significant way to the greater research community. My internship mentors, Arch Robison, Lee Baugh and Robert Knight also contributed in making me discover the research world in industry, a direction I ended up choosing for my current career. I would also like to thank my committee Hyesoon Kim, Karsten Schwan, Umakishore Ramachandran and Sudhakar Yalamanchili for their help and feedback with greatly improved this thesis.

Finally, I would also like to thank all the support staff at Georgia Tech who worked hard to simplify the complex administrative apparatus of Georgia Tech. In particular I would like to thank Susie McClain, Deborah Mitchell and Della Phinisee.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Whereas earlier generations of computers were resource-scarce, modern multi-core and many-core machines are resource-rich. Historically, software optimizations were geared towards "fitting" the computation inside scarce resources whereas modern multi-core machines face the dual problem of idling resources on the one hand and sequential bottlenecks on the other. The *opportunistic computing* paradigm, on which this thesis rests, is the idea that the computation (sequential or otherwise) should dynamically scale to occupy idling resources to enhance its speed or quality thereby solving both problems.

This thesis focuses specifically on *hard to parallelize* computations which cannot easily scale to occupy more and more resources. We have observed that traditional data and task parallelism do not allow these types of computations to scale as this type of parallelism is either hard to express (for algorithms that have unstructured memory access patterns for example) or does not exist (for purely sequential computations). We instead propose to exploit the *algorithmic properties* of a computation to develop programming models that utilizes parallel resources to improve performance or quality for hard to parallelize computations. Specifically, this thesis looks at three distinct algorithmic properties: **i)** algorithmic diversity, **ii)** the semantic content of data-structures, and **iii)** the variable nature of results in certain computations.

Our first contribution is the N-way programming model which exploits algorithmic diversity to opportunistically speed-up or improve the quality of a computation. The N-way model is specifically tailored for sequential computations, providing speedup for such computations and thereby providing a solution to the bottleneck expressed in Amdahl's law. The N-way model relies on the fact that for many problems, multiple

ways exist to solve them, each potentially differing in their expected completion time, resource requirement or quality of result.

An intuitive example of algorithmic diversity is a randomized algorithm: each launch of the algorithm on a given input will behave differently. The N-way model launches multiple competing ways performing the same computation and picking the best one (fastest or best quality) just in time. The more diversity exists in a problem, the greater the speedup or QoR improvement potential is. It is important to note that the amount of diversity in a problem is not dependent on the sequential or parallel nature of the algorithm used to solve the problem; in other words, N-way parallelism is equally applicable to parallel and sequential codes.

This thesis also develops ways to minimize the number of ways launched (n) while maximizing the probability of improvement. Indeed, the N-way model can be very wasteful as only one of the ways ends up successfully "committing" its result. The N-way system attempts to solve this problem by developing **i)** a statistical learning approach which estimates the benefit of different amounts of speculation and **ii)** a mechanism to reclaim unproductive ways to further reduce n during execution of the competing ways. Both these techniques allow N-way to maximize the benefit obtained while minimizing the amount of resources required. Through the use of N-way model, we show very high (super-linear) speedups on hard to parallelize combinatorial problems such as SAT solving.

Our second contribution is an extension of the N-way model allowing for additive semantics instead of purely competing semantics: optional additional ways can be used to improve the quality of result in a main thread when they are joined back into it. Indeed, for many applications, particularly in the gaming and multimedia domain, multiple results are "correct" although some are better than others in terms of quality. Additional, quality enhancing ways, can therefore be launched and, if time and resource permits it, their results can be merged back into a main thread.

Finally, we present a framework to improve optimistic parallelism by leveraging the semantics of data-structures and algorithmic properties to dynamically predict conflicts and reduce the overhead of optimistic parallelism such as Software Transactional Memories (STMs). Indeed, while optimistic parallelism techniques have been shown to be beneficial in writing parallel versions of hard to parallelize algorithms (such as algorithms relying on sparse and irregular data-structures with hard to discern patterns), the overhead of wrong predictions can be very high. The technique we present allows the programmer to specify semantics information concerning the data-structures with the help of *predicates* that can guide an optimistic runtime in making the correct decisions to minimize wrong predictions. We further develop a profiling-based approach to automatically determine the symbolic data footprint of a transaction which permits the automatic generation of conflict prediction functions.

CHAPTER I

INTRODUCTION

Today, Moore's law, which states that the number of transistors in a chip doubles approximately every two years, is causing an over abundance in computing resources due to the fact that transistors are now being used to create more and more cores. In desktops and laptops, multi-cores have been present for years: the Westmere family of processors comes with 6 cores and the Haswell [37] family of processors is rumored to come with 8 cores standard. Intel has also introduced the SCC ("Rock Creek") architecture which provides 24 dual cores on a single chip [38]. Mobile devices have also recently gained multi-core processors like the ARM Cortex-A9 [4]. As the number of cores increases, applications must learn to harness this new found power in non-traditional ways.

Prior to multi-cores, applications could freely benefit from the sequential execution improvements that came with frequency scaling. This is no longer the case and although parallel hardware confers, in theory, vastly greater computing power to end-user applications, it is now up to the application to effectively occupy the hardware. Certain applications, such as those used in the high-performance computing (HPC) domain, are embarrassingly parallel and have no trouble scaling with the increasing number of cores. Although the *effective* and *efficient* exploitation of massive parallel resources poses certain problems (for example, code and data locality issues), fundamentally, the applications can scale to more and more cores. Other applications, however, cannot readily occupy parallel hardware.

1.1 A brief history of multi-cores

Although parallel resources have been available for a long time, the phenomenon of multi-cores is fairly recent. Sun introduced a 64 bit dual-core processor (the UltraSPARC IV) in 2003 and Intel waited until 2006 to introduce the dual-core Core Duo. Since then, the number of cores on a die has gone up exponentially. Intel introduced its first quad-core (Kentsfield) at the end of 2006 and a six-core in 2008 (Dunnington). In 2006, Sun also introduced the UltraSPARC T1, an 8-core processor. The latest generation of Intel chips, the Sandybridge Core i7 are slated to be released in a 6-core version. Intel also recently presented Rock Creek [38] which is dubbed a 'datacenter on chip' and sports 48 cores. The trend is thus clearly towards more and more parallel resources: multi-cores are here to stay.

When dual-cores first came out, operating systems could occupy both cores relatively easily. However, as the number of cores increase, fully occupying these cores is becoming more and more of a challenge: not only are workloads hard to parallelize but desktop users frequently do not actively run multiple applications at once thereby restricting the number of parallel processes that need to run. Furthermore, workloads and tasks that are easily parallelizable are being offloaded it to accelerators such as GPUs further leaving the CPU cores unused.

1.2 Road-blocks to parallelization

While embarrassingly parallel applications are efficiently parallelizable, other types of applications have more difficulty in exploiting parallel resources:

Applications with 'may' dependencies Certain applications contain natural parallelism but this parallelism is difficult to express due a dynamic structure of data dependencies. This is particularly true for applications which utilize trees or other pointer-based structures as their underlying data-structure. For these

applications, there *may* exist a data dependency that makes parallelization impossible but not always. Determining the cases when parallelization is possible is a difficult problem and therefore, expressing and exploiting parallelism in such applications is not an easy task.

Applications with 'must' dependencies (sequential) Other applications simply do not contain "break-up" semantics on which data and task parallelism rely: a dependency prevents breaking-up of the computation. For these applications, data and task parallelization techniques are not applicable.

In this section, both types of applications are analyzed.

1.2.1 'May' dependencies

Irregular algorithms are defined as those that rely heavily on pointer-based data structures such as graphs or trees (the STAMP benchmarks for example [13]). An important characteristic of these algorithms is that the exact elements, and therefore memory location, they access are heavily data-dependent and cannot be known until runtime. This cripples potential static analysis such as those used to efficiently parallelize dense matrix computations in the HPC domain. However, these algorithms can still benefit from parallelization [58]. They are also increasingly present in emerging domains [63] such as machine learning, social network analysis an event-driven simulations.

1.2.1.1 Differences with regular parallelism

For the purpose of this section, a computation is defined as being composed of an operation and a data footprint. This view was also taken by Mendez-Lojo et al. in [55] where they stress the importance of a *data-centric* view of the computation. For an operation O, its data footprint will be noted $D_f(O)$.

In a typical HPC application, the operations execute on a dense matrix. Computing $D_f(O)$ is therefore made easy by the fact that a matrix is completely indexable. In many cases, a compiler can reason about the indices involved for each operation and statically compute $D_f(O)$.

In an irregular application, however, $D_f(O)$ can only be computed with knowledge of the *runtime values* of variables which are not as easily bound as indices. Furthermore, the use of pointers complicates analysis even further as the variable from which to derive a value may be only indirectly accessible through the runtime value of another variable.

1.2.2 The bottleneck of 'must' dependencies (sequential code)

In both previous cases (regular and irregular parallelism), current parallelization approaches (data/task parallelism) rely on *breaking-up* a computation into pieces which can be processed in parallel. This process, however, has its limits as noted by Patterson in [62]. In the extreme case (for algorithms that have a very sequential execution), this limit will be one and the code cannot be broken up into individual pieces that can be processed in parallel.

1.2.2.1 Inherently sequential codes

Furthermore, certain problems are known to be "inherently sequential" which means that traditional parallelization cannot effectively speed them up. Indeed, in complexity theory, two classes in particular are key to determining problems which cannot be solved efficiently on a parallel machine:

The `NC` complexity class The `NC` complexity class[1] is the class of problems that can be solved in polylogarithmic time on a parallel computer with a polynomial number of processors. In other words, there exists constants a and b such that

[1] `NC` stands for "Nick's Class" after Nick Pippenger who did extensive research on the subject.

any problem P in NC can be solved in $\mathcal{O}\left(\log\left(n\right)^a\right)$ on $\mathcal{O}\left(n^b\right)$ processors.

The P-complete complexity class The P-complete complexity class is the class of problems in P[2] such that every other problem in P can be reduced to it.

Trivially NC $\subseteq$ P as parallel computers can be simulated on a sequential machine. However, it is not known but widely suspected that NC $\neq$ P. If this supposition is indeed true, it means that any problem in P-complete is not in NC (if one considers NC reductions) and are therefore not effectively parallelizable.

Therefore, for P-complete problems, a list of which can be found in [29], traditional parallelism is only of limited use [30]. Alternative approaches to effectively parallelizing these and other applications are thus required. Some example problems (taken from [29] include:

- Graph search algorithms such as breadth-first search (determining if a node v is visited before a node u in such a search).
- Graph theory algorithms such as the nearest neighbor traveling salesman heuristic.
- Combinatorial optimizations such as linear programming.

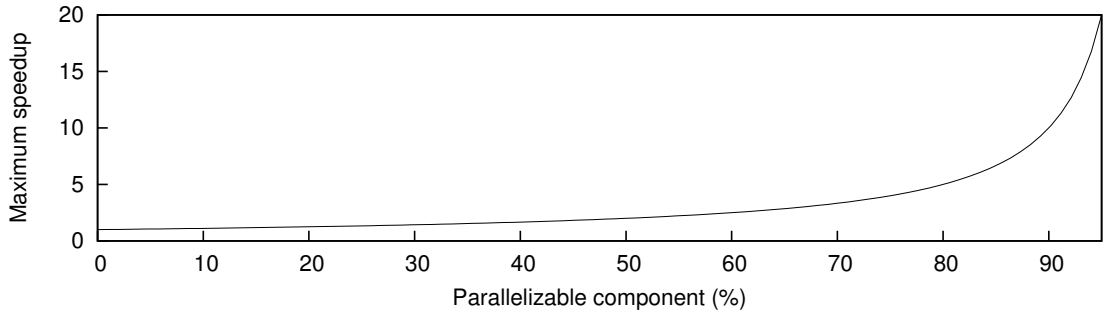
One can see that these problems are fundamental in emerging areas such as search, social networking, etc. It is thus crucial to address the fundamental issue that these algorithms cannot be effectively parallelized.

1.2.3 Importance of dealing with hard to parallelize codes

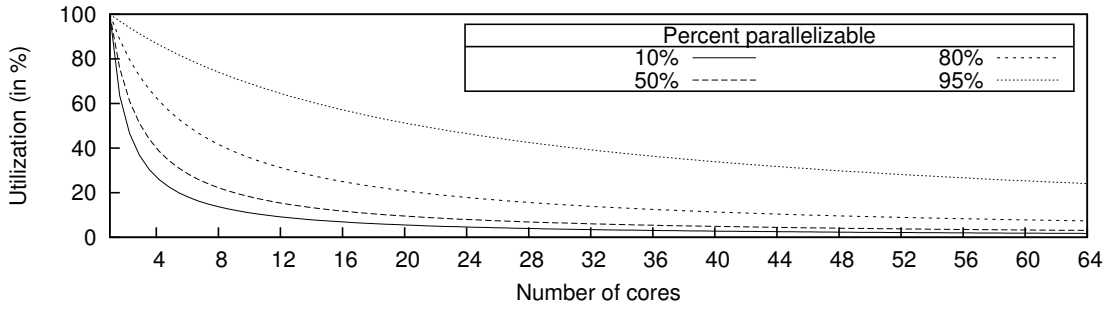
Dealing with sequential code is crucial if one wants applications to keep scaling in performance as illustrated by Figures 1. Figure 1(a) shows the maximum achievable speedup as a function of the amount of parallelism in an application (Amdahl's law)

[2]Class of problems that can be solved in polynomial time.

and Figure 1(b) shows the maximum achievable core occupation as a function of the number of cores (assuming that the parallel sections of code are embarrassingly so). Both figures show that sequential, or even hard to parallelize, components in



(a) Amdahl's law



(b) Achievable core occupation: $U(N) = \frac{1}{P+N(1-P)}$

Figure 1: Motivating trends

applications not only limit speedup but also cause more cores to idle thereby wasting precious resources. In [35], Hill studied Amdahl's law and came to similar conclusions.

1.3 Current solutions

Currently, three automated families of techniques exist to improve the performance of hard to parallelize algorithms.

1.3.1 Frequency scaling

Frequency scaling relies on the intrinsic improvement of the speed of the underlying hardware. Until the advent of multi-core processors, this was the main driving force in the huge performance improvements between the early 1970s and the mid 2000s.

In Figure 2, it is clearly evident that the increase in processor frequency slows dra-

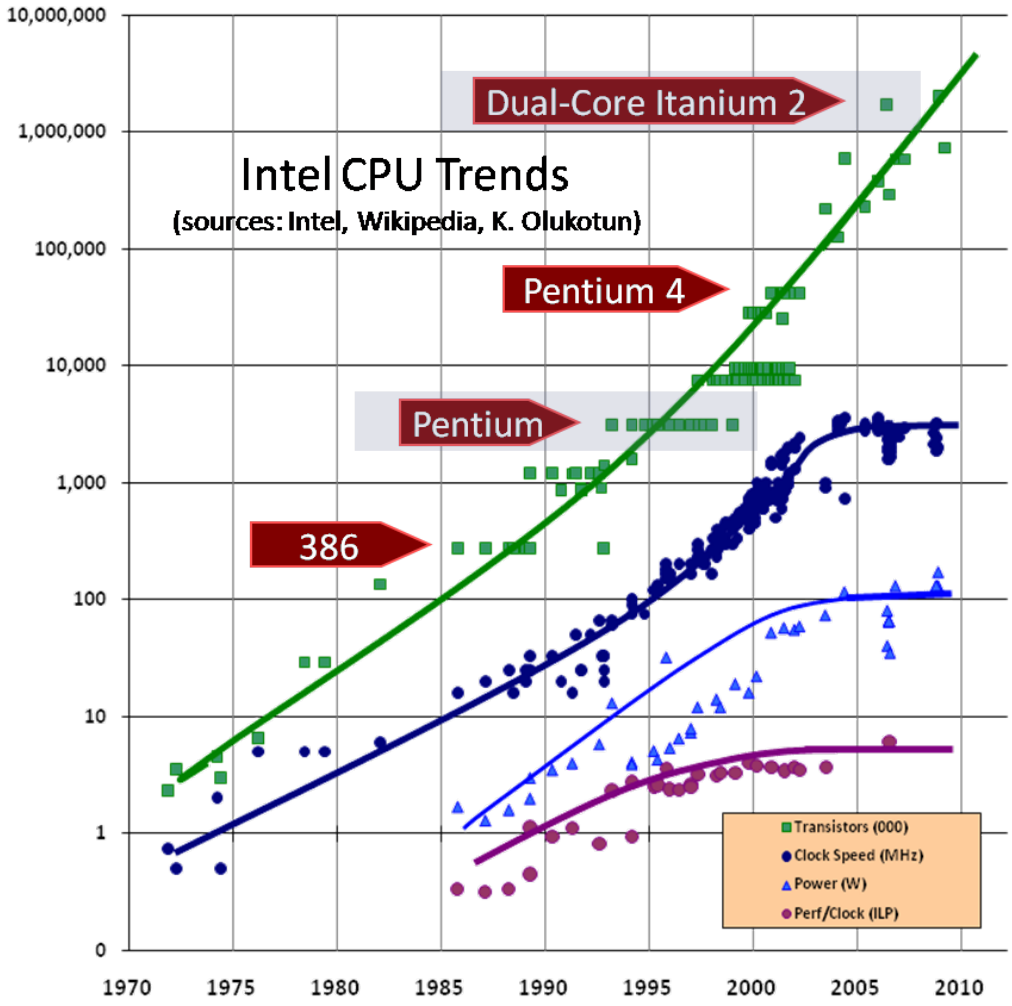


Figure 2: Evolution of CPU speeds from the 1970s to today (taken from [73])

matically in 2005 and has mostly stagnated since then. For desktops and laptops computers, the benefits of this solution will therefore be marginal. Mobile devices may benefit from it as processors found in mobile devices such as phones are still running much slower than their desktop counterparts mostly due to power and heat issues.

1.3.2 Speculative and optimistic execution

Speculative execution can be broadly defined as doing work that may or may not be needed. This has been exploited at the hardware level for a long time. For example, the result of branches are typically predicted and the most likely branch is speculatively executed until a definite answer on the branch condition is evaluated. If the prediction turned out to be accurate, the execution continues as usual; if not, the execution is canceled and restarted at the beginning of the other branch. Speculative execution proved useful in hardware and the idea migrated to software. Uses of speculative execution include "helper" threads [72, 85]. Helper threads are used to assist the main threads, for example to prefetch certain memory locations. The locations prefetched are not accurately known but the helper threads speculate in order to potentially speed up the execution time of the main thread. Another example is thread-level speculation [61] which is used to execute multiple paths of execution simultaneously.

Optimistic execution is slightly different and consists of executing a needed computation in parallel with other computations without knowing if the parallelism is legal. If the execution turns out to be legal, parallel forward progress has been made. The most popular use of optimistic execution is transactional memories (hardware based, also known as HTMs or software based, known as STMs). TMs systems allow the programmer to define sections of code which are to be executed "atomically". Such a section of code, also called a transaction, is deemed to have executed atomically if no 'write' operation occurred on any of the data it touched during its execution. A runtime system ensures the atomicity property of transactions by speculatively executing the transactions and only committing their results if the atomic property was satisfied. If it was not, the transaction is canceled (rolled back) and re-executed. TMs do not help in the case of sequential codes but make the expression of parallelism in irregular algorithms much simpler as the programmer does not have to explicitly

worry about overlapping data footprints.

Note that in this thesis we do exploit the ideas of speculative and optimistic execution but in ways that are different from current ones.

1.3.3 Discovering dynamic parallelism

Programming models have also been developed which help a runtime dynamically discover parallelism instead of a relying on a static expression of data or task parallelism. One such model is the Concurrent Collections (CnC) model [44, 11] developed by Intel (based on TStreams originally developed by Kath Knobe). In CnC, programmers specify the inputs and outputs for each computational step and a runtime dynamically launches the computational steps when possible. Therefore, the programmer does not explicitly specify any specific form of parallelism and it dynamically appears. Note that CnC is part of a larger family of very similar programming models. CnC finds its origins in TStreams which has its origins in Stampede [60, 64]. Later, Mandviwala, in [54] developed Capsules which also addresses the issue of discovering dynamic parallelism but seeks to dynamically, and at runtime, adapt the level of granularity of the parallel computation thereby trying to improve the efficiency of the parallelization.

1.3.4 Limitations

The above techniques have several limitations:

- The fundamental problem behind sequential algorithms is not addressed. Indeed, prefetching helper threads, while useful, will not scale to an increasing number of cores. TMs and the dynamic discovery of parallelism is not applicable to sequential algorithms.

- Speculative or optimistic execution has two major pitfalls. Firstly, errors in speculation can be very expensive as they usually involve the canceling of past execution and a rollback to a previous "safe" state. Secondly, it becomes harder

and harder to accurately and correctly speculate. In the case of optimistic execution, the greater the number of concurrent transactions, the greater the chance of a conflict.

This thesis seeks to address these limitations by exploiting *algorithmic properties* and provide scalable programming models that allow both sequential and irregularly parallel applications to efficiently exploit multiple cores.

1.3.5 Approach: novel programming models

In this thesis, novel programming models are proposed to exploit idling parallel resources and improve computations irrespective of their sequential or parallel nature. The goal is to **i)** better occupy the plethora of resources that are available and **ii)** provide speedup or quality improvements to computations.

This thesis builds on the ideas of *opportunistic computing* expressed in [16]. Opportunistic computing is the idea that programs should dynamically take advantage of the opportunities offered to them, either in terms of increased resource availability or increased time to produce a result. This thesis specifically focuses on effectively utilizing idling and wasting parallel resources to improve the performance of sequential codes as well as irregular parallel codes.

1.4 Exploiting algorithmic properties

Current parallelism approaches all rely on a breaking up of the computation, either through the expression of data parallelism or task parallelism. As discussed earlier, this approach is incompatible with purely sequential algorithms. This thesis proposes to exploit *algorithmic properties* to open up new avenues of parallelism or to improve existing ones.

An algorithm involves the execution of a computation on an underlying data-structure to produce a result. Each of these elements has algorithmic properties that can be exploited:

Property of the computation *Algorithmic diversity* which is defined as the existence of multiple "ways" to solve a given computation can be exploited. For many important problems, different ways to solve them may exist and exhibit *varying execution times even on the same input.* A runtime can, if resources are available, launch multiple such ways in parallel and pick the fastest in a just-in-time fashion. This will lead, on average, to a speedup.

Property of the data-structure Data-structures typically have a lot of *semantic* information associated with them: the programmer assigns a meaning to each of the elements in a data-structure that is specific to the way the data-structure is being used in the computation. These meanings are however lost to the compiler but precious information about the data footprint of operations can be gained from this semantic knowledge. Allowing the programmer to provide a runtime system with the opportunity to exploit this information can improve the performance of optimistic parallelism by reducing the probability of rollbacks.

Property of the result Many emerging applications, such as those in the gaming or multimedia area, are amenable to *variable semantics*: for a given computation, there can be multiple *correct* solutions, some potentially better than others. This property can be leveraged by opportunistically launching, in parallel, quality improving computations. This allows an application to dynamically adapt to the dynamic resource envelope.

Each aspect is described in more detail in the following sections.

1.4.1 The N-way model: exploiting algorithmic diversity

The N-way model exploits an algorithmic property, diversity, which is orthogonal to the properties exploited in the traditional 'breaking-up' parallelism. N-way thus enables speedups on hitherto *sequential* computations as the traditional distinction

between 'sequential' and 'parallel' computations no longer applies in N-way: the distinction is now between computations which exhibit diversity and those that do not irrespective of whether or not the computation can be broken up into independent tasks and/or data chunks.

Diversity is naturally present in many complex problems: NP-hard problems are computationally intractable to solve exactly and therefore have a plethora of approximations and trade-off based algorithms which are used to solve them, thereby contributing to making such problems diverse. Randomized algorithms, which execute differently on the same input, also lead to diversity within a problem.

The N-way model seeks to exploit this diversity by launching, in parallel and in isolation, the diverse ways that can be used to solve a problem. A runtime can then pick, just-in-time, the best execution way (for example the fastest) and discard the others. Note that the same framework can also be used to obtain quality of result (QoR) improvements by picking the way with the best QoR instead of the fastest one. While this approach will, in theory, provide an expected speedup, in practice, the extra resources consumed by the competing ways may diminish the potential theoretical speedup. Therefore, this thesis develops a N-way model that incorporates a mechanism to try to predict the optimal set of ways to launch to maximize speedup with respect to the number of resources used.

This thesis presents a full framework capable of expressing and exploiting algorithmic diversity. The proposed framework is capable of extracting speedups in hitherto sequential computations.

1.4.2 Determining semantic data footprints

As previously discussed, an important consideration when parallelizing operations is the correct determination of their data footprints ($D_f(O)$). Irregular algorithms lack apparent static dependence structures which makes this determination difficult

if not impossible. However, these algorithms frequently exhibit *semantic structure* where the programmer can frequently *estimate* $D_f(O)$ based on very limited knowledge (the input to the operation for example). Through the estimate of $D_f(O)$, a runtime system can then make judicial decisions regarding the optimistic scheduling of transactions by ensuring that the probability of conflict between two concurrently running transactions is low.

This thesis presents:

- A profiling-based tool to automatically understand and construct the semantic data footprints of operations.
- A framework capable of exploiting the semantic knowledge of data footprints to throttle transactions in an optimistic system.

1.4.3 Quality driven computing: exploiting variable semantics

The N-way model presented in Section 1.4.1 launches multiple competing ways to gain speedup QoR improvements. However, for applications amenable to variable semantics, it is also possible to launch collaborating ways instead of competing ones. In games, for example, the artificial intelligence (AI) entities that operate certain elements of the game can be of varying quality. More realistic effects can be added to make the game appear closer to reality. As an illustration, a more precise modeling of the human body can be used to calculate how a character moves down stairs (in most games, the feet "hang" in the air, more precise calculation can make this effect go away). In video coding, the way in which one encodes an image is variable; for example, the MPEG format has three types of frames (I, P, or B) [27]. The percentage of use of each of these types of frames can result in variations with respect to the encoded size and decoding time. Given more resources, higher quality and more interesting processing can be done as a part of these applications' semantics.

This extension to the N-way programming model seeks to exploit this variability in semantics by launching in parallel optional quality-enhancing computations that, if given sufficient resources and time, can enhance the final result of the main application.

1.5 Thesis statement

The time when computations had to be fitted to the hardware is long gone and programmers now face the dual problem of occupying multiple parallel resources and dealing with the bottleneck of hard to parallelize or sequential applications. For these computations, traditional parallelization techniques such as data and task parallelism are not applicable or inefficient. This thesis proposes that *algorithmic properties* can be exploited to provide new avenues to simultaneously solve the problem of efficiently occupying cores and improving sequential or hard to parallelize computations. This thesis will show that significant improvements in the execution time or quality of result of computations can be achieved though a better understanding and exploitation of these properties. Specifically, this thesis explores three algorithmic properties: algorithmic diversity, data-structure semantics and variable result semantics.

1.5.1 Contributions

This thesis makes the following contributions:

- Recognizing that algorithmic properties are under utilized in today's programming models and proposing to exploit three specific algorithmic properties: diversity in computations, semantics of data structure and variable semantics of results.

- A *programming model* called N-way parallelism which enables a non-traditional speculative parallelization of sequential problems through the exploitation of

diversity in computations. This thesis further details a runtime that automatically manages the parallel efficiency of the N-way model.

- A framework enabling the expression of the semantics of data-structures. This thesis further describes a runtime capable of exploiting such information and improving the performance of optimistic parallel codes. Finally, this thesis also explores the possibility of automatically extracting the semantics of the data-footprint through a profiling approach.

- A quality-driven *programming model* enabling the programmer to design an application exhibiting variable semantics so that the application dynamically adapts to fully occupy the available resources while still meeting quality constraints.

CHAPTER II

EXPLOITING ALGORITHMIC DIVERSITY THROUGH THE N-WAY MODEL

The N-way model is designed to address the bottleneck of sequential codes as described in Section 1.2.2: it aims to exploit parallel hardware and provide speedup to hitherto sequential code. The N-way model therefore is a solution to the limitations imposed by Amdahl's law.

2.1 Introduction

The N-way model relies on the observation that for many important computations, there is a multitude of ways to solve them which can lead to *varying execution times even on the same input*. This diversity can come from the use of heuristics or randomness in algorithms. Therefore running multiple *ways* in parallel and picking the fastest will provide an *expected* speedup. Complex problems such as NP-hard problems are computationally intractable to solve exactly and therefore have a plethora of approximations and trade-off based algorithms that can be used to solve them. Randomized algorithms, which execute differently for the same input also lead to diversity which can be exploited. Section 2.2 elaborates on the presence of diversity in many important problems.

2.1.1 An alternative to 'break-up' parallelism

The attractiveness of the N-way model resides in the fact that it exploits an algorithmic property, diversity, which is orthogonal to the properties exploited in the traditional 'breaking-up' parallelism. N-way thus enables speedups on hitherto *sequential* computations as the traditional distinction between 'sequential' and 'parallel'

computations no longer applies in N-way: the distinction is now between computations which exhibit diversity and those that do not irrespective of whether or not the computation can be broken up into independent tasks and/or data chunks.

Therefore, the speedup provided by N-way *does not* come from parallelism over multiple cores. The speedup provided comes from improving the *expected completion time* of a computation over n cores as opposed to the expected completion time over a single core. The potential speedup for N-way is therefore directly related to the amount of *diversity* present in the ways to solve the computation. In particular, if the various ways to solve the computation C exhibit very little diversity in their execution time, N-way will not be able to provide speedup irrespective of the number of parallel resources committed. Conversely, if a large spread of execution time exists, N-way will provide a speedup which may be super-linear.

Note that the *expected* completion time of a computation is considered as the N-way approach is a probabilistic one. Consider for example a computation C implemented as a randomized algorithm R which is run on an input I. Since the execution time of $R(I)$ may vary from run to run (due to its random nature), an expected completion time for C on input I must be computed: it corresponds to the *average* execution time of $R(I)$ over multiple runs. Similarly, suppose that C is now implemented using various heuristics H_1 to H_h. Since each heuristic executes differently, the expected completion time for C is again the average of the completion times of H_1 to H_h on input I.

2.1.1.1 A simpler model

Contrary to other parallel models, the N-way model is free from complex and error-prone parallel programming constructs such as threads, locks and barriers. As far as the programmer is concerned, each way is a separate sequential program that can be written using sequential techniques. Figure 3 illustrates this: B is a step in

the problem that can be solved with multiple ways. The N-way model guarantees semantic equivalence between 3(a) and 3(b).

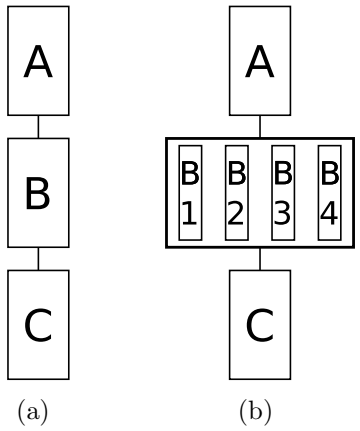


Figure 3: Sequential flow versus N-way flow

2.1.2 Example

A very simple example of a N-way computation is the problem of finding a path between two nodes in a graph. Certain algorithms will perform an exhaustive expanding search (such as Lee's algorithm), others will perform a random search of the graph and yet others use heuristics to approximate the best next point (the A^* algorithm for example). Therefore for a single computation (finding a path), multiple *ways* exist to solve it, each giving a valid solution in potentially varying amounts of time for the same input. Ideally, for each input to the problem, an oracle could determine the best (fastest) algorithm to run for the given platform. However, such an oracle would most likely be as computationally intensive as the algorithms themselves. The N-way model solves this problem by removing the obligation to choose a-priori the way which will execute the fastest, preferring instead a just-in-time solution where the fastest way is picked when it completes. The execution time of the problem is thus determined only by that of the fastest way, all other ways being discarded.

2.1.3 Problem: a potentially wasteful model

The N-way model is potentially wasteful of resources. Indeed, if n ways are launched on n cores, $n-1$ cores will perform work that will eventually be discarded. Increased waste can lead to an increase in memory and bandwidth contention therefore limiting or negatively impacting the speedup obtained. For N-way to be practical in a wide range of scenarios, it is thus crucial to answer the following question: *how many and which ways to launch in order to best balance speedup and resource usage*? In other words, it is important to determine how the expected speedup *scales* with resources. Note that *the notion of scaling is the opposite of what is traditionally considered*: one wants to determine the *minimum* n that best balances expected speedup and resources used. The optimal n will be directly related to the amount of expressible diversity: in an extreme case, if all ways solve the problem in exactly the same execution time and quality, n should be 1 as no benefit will be gained from launching multiple ways.

Answering the previous question is difficult for the programmer because of **i)** the dependence of kernel execution time on input data and **ii)** the lack of models linking execution time of the various ways together.

This thesis develops a runtime system capable of *learning* the execution time characteristics of the N-way computation (C) based on prior invocations thereby enabling the runtime to *estimate* the benefits of launching various sets of ways in subsequent invocations of C. If C is not invoked multiple times, N-way is still applicable but not the learning model.

Even if the optimal set of ways is picked, some ways will make *progress* towards C's solution much slower than others making them highly unlikely to complete first. Additional resource savings can therefore be had by killing these unproductive ways early. For many applications, it is straightforward for the programmer to define *progress monitors* which can be monitored by the N-way runtime in deciding which ways to kill. This technique, which is called *culling*, allows significant additional

resource savings when the programmer can define the progress metrics.

Therefore, the learning of n and culling are two independent techniques which can be applied individually or together for even more saving of resources.

2.1.4 Terminology

Throughout this chapter, the following terms are used:

- **N-way computation** or 'computation' refers to an algorithmic step to which the N-way technique can be applied. In other words, a N-way computation can be executed in a variety of "ways" for a given input and each way will produce a solution to the computation.

- **N-way kernel** or 'kernel' refers to a specific implementation used to solve a N-way computation. For example, if multiple heuristics are used to solve a N-way computation, each heuristic will be called a N-way kernel.

- **Expected speedup** refers to the speedup that N-way will provide. It will be noted S_n for n cores.

- **Effective utilization** or 'utilization' refers to the effective number of cores utilized by a N-way computation. This is noted n_{eff}. $n_{eff} \leq n$ where n is the number of cores allocated to the N-way computation. n_{eff} can be smaller than n in the presence of culling for example when some cores are freed before the completion of the computation.

2.2 Diversity

The N-way model relies on the existence of *diversity* in common computations. This section motivates the presence of such diversity in many applications.

Consider a N-way computation C. By definition, C is a diverse computation that can be solved using a multitude of ways. The different ways may solve C differently,

some operating faster than others or with a higher quality of result (QoR). Note that the granularity of C is irrelevant in the definition. C could be an entire application or a small basic building-block kernel such as a "sort" problem. Finding diversity at the granularity of a kernel will allow larger problems that depend on the kernel to also benefit from the diversity present in the kernel.

2.2.1 Algorithmic diversity

Three types of algorithmic diversity in a computation C can be identified: **i)** across distinct algorithms, **ii)** within an algorithm if it can be parametrized and **iii)** within an algorithm if it utilizes randomness.

Diversity across algorithms For many real-world problems, such as NP-hard problems, finding an exact solution in a reasonable amount of time is impossible. Even for problems in P, the large size of the problem may make exactly solving it problematic. For such problems an *acceptable* solution rather than an optimal one is sought. For example, when finding a path between two nodes in a graph, any path is an acceptable solution although some may be *preferred* over others (shorter paths for example). Acceptance of a wider set of solutions enables the use of a variety of algorithms to solve the same problem. Approximation algorithms [81] have been utilized for such a purpose: they give a solution of provable quality within provable runtime bounds[1]. Similar to approximation algorithms, heuristics based algorithms provide the same notion of diversity without provable quality and/or runtime bounds.

Diversity in parametrized algorithms Apart from distinct algorithms, even if only a single algorithm exists to solve C, diversity may still be present if the algorithm can be parametrized. For example, the CPLEX solver [75] offers over 100 parameters

[1]Note that the bounds are worse-case bounds and not precise enough to determine which algorithm will perform better on a specific input.

to tune its algorithm. Each pair (A, Param) can be considered a distinct algorithm with differing execution characteristics as in the previous case.

Diversity due to randomness Randomized algorithms [57, 56] utilize a degree of randomness as part of their logic. A randomized algorithm seeks to achieve good performance on average. Due to its random nature however, its running time, its output, or both are probabilistically characterized random variables.

Randomized algorithms are therefore parametrized implicitly by the random seed used. The difference with parametrized algorithms is that the parameter space is innumerable (for all practical purposes). Note also that randomized algorithms are particularly interesting for diversity as a single algorithm will provide diversity as opposed to different algorithms in the other cases.

2.2.2 Other sources of diversity

While this work focuses on the exploitation of algorithmic diversity, diversity also occurs at the hardware level (due to architectural heterogeneity for example) and at the compiler level (different optimization options). This latter aspect was explored in more detail in [78]. These sources of diversity only enhance the current ones identified and described.

2.2.3 Diversity is common

This section has shown that different types of algorithms commonly used to solve difficult problems contain diversity which can be exploited. Asanovic et al. describe 13 dwarfs [5] as forming the cornerstone of tomorrow's computation. Diversity is present in many of them:

- **Linear algebra (dense or sparse)** Parametrized algorithms are frequently used. Certain randomized algorithms are also used [79].

- **Combinatorial logic** Given their high computational complexity, such problems are frequently approximated [81]. Randomized algorithms are frequently used and this chapter presents results with WalkSAT [71] a randomized SAT solver.

- **Graph traversal** Randomness is frequently utilized to solve large-scale graph problems given their good average case complexity. In particular, many AI algorithms utilize graph traversals to evaluate possible "moves". This chapter presents results for a randomized Hamiltonian cycle builder.

- **Graphical models** Searching for relationships in Bayesian networks is a NP-hard problem and many different search algorithms can be applied. Non-exhaustive searches frequently make use of heuristic and random exploration as part of their algorithm. Simulated annealing and genetic algorithms are examples of that.

- **Backtrack / branch-and-bound** The Traveling Salesman Problem (TSP) is a classical example of a branch-and-bound algorithm. Quickly exploring the huge space of possibilities requires the use of heuristics and/or randomness. This chapter presents results for heuristical TSP solvers.

2.3 N-way model

The N-way model seeks to exploit the diversity described in Section 2.2 to provide both expected speedup and QoR improvements; for clarity, this section focuses on expected speedup and briefly describes how QoR can be provided in Section 2.3.3.

2.3.1 Base model

As shown in Figure 3, the N-way model exploits the diversity in B to transform the sequential execution flow shown in Figure 3(a) to the parallel one shown in 3(b) in a way that makes both flows semantically equivalent. If we assume that each way

returns a valid solution, semantic equivalence is ensured by the enforcement of the following two rules:

- Parallel ways execute in isolation from one another in a side-effect free manner
- One and only one of the parallel ways makes its computation visible to the rest of the program. In Figure 3(b), C will only be affected by one and only one of B_1 to B_3

Apart from ensuring semantic equivalence, isolation (described in Section 2.5) also enables the programmer to program each way as if it were sequential code. While B_1, B_2 and B_3 all execute in parallel, C will only be affected by the execution of one and only one.

While the idea behind N-way parallelism is simple and previous work [16, 78, 77] has shown that naively launching as many ways as possible maximizes expressed diversity and therefore expected speedup potential, the real difficulty lies in determining the best balance between resources used and expected speedup potential to make N-way efficient, practical and scalable. Launching too few ways will reduce speedup potential while launching too many will waste resources and energy as well as possibly degrade the performance of the ways thereby leading to sub-optimal speedups. Determining which and how many ways to launch is dependent on **i)** the inputs because of the data-dependent behavior of many algorithms, and **ii)** the characteristics of the algorithms themselves, for example the execution time distribution of the randomized algorithms or the applicability of a particular heuristic to a given input. The information required is thus not easily available to the programmer making it difficult for him to a-priori make an informed decision. In this section, a statistical learning model that can effectively determine the best combination of ways to launch is described.

2.3.2 Efficient N-way model

In traditional parallelism, the number of parallel resources that can be effectively utilized is bounded by the amount of divisibility (work or data) in the algorithm. N-way exploits diversity instead of divisibility and therefore the number of parallel resources that can be effectively utilized is limited by the amount of diversity present in the computation. Intuitively, for a given input, if the *spread* of completion times of the different ways is wide (large diversity), many resources can be effectively used to "explore" this spread whereas if the completion times fall in a narrow band, allocating more and more resources will only provide a marginal benefit. As an extreme example, a computation which can only be solved using a deterministic algorithm exhibits no diversity (a spread of 0) and utilizing more than one resource to solve it is wasteful.

While models exist to a-priori determine the potential speedup that can be obtained through the division of work, the amount of diversity in algorithms is difficult to characterize statically as it is heavily data-dependent in a non predictable manner. An exact determination of the number of resources needed to optimally express diversity in a problem is therefore not possible and we instead propose a statistical model which allows a runtime to compute the expected speedup associated with a particular choice of ways. The expected *parallel efficiency* defined as $\mathrm{P_{Eff}}(n) = S_n/n$, where S_n is the expected speedup on n resources, can be computed and maximized. Note that the expected parallel efficiency is computed as a function of n. Culling may reduce the utilization of cores from n to n_{eff}.

2.3.2.1 Assumptions

N-way does not rely on the programmer to provide information about the computation C's execution-time distribution. Instead, the following assumptions about how the application invokes C are made:

- **Repetition**: The N-way computation C is invoked repeatedly within an application.

- **Stability**: The underlying (unknown) execution-time *distribution* for C only changes *slowly* over consecutive invocations of C.

These assumptions allow the construction of a statistical learning scheme that can reliably estimate C's execution time distribution based on previously observed behavior over past inputs.

Stability assumption Note that the stability condition *does not* require that consecutive inputs be similar to one another but rather that the behaviors of the various ways observed over previous inputs remain stable. For example, consider a randomized algorithm being fed two different types of inputs, one large where the randomized algorithm takes a fairly long time to return a result and one small where the algorithm returns quickly. Even if individually the large and small inputs are very different, if future inputs are similar to either the large or the small ones, the execution time distribution of the randomized algorithm will not change: it will be the superposition of the execution time distribution of the small input and that of the large input.

Similarly, consider for example that the execution time distribution is originally a Gaussian distribution $\mathcal{N}(m, s)$. If N samples (execution times) are drawn from this distribution, they are most definitely *not* equal or even similar. Collectively, however, they form $\mathcal{N}(m, s)$. If the distribution now changes to $\mathcal{N}(m_1, s_1)$ and M samples are drawn, the M samples will collectively indicate the $\mathcal{N}(m_1, s_1)$ distribution. Moreover, a mix of samples (from the first N and from the last M) will indicate another distribution but if the change from m to m_1 and s to s_1 is small, the mix will closely approximate $\mathcal{N}(m_1, s_1)$. The learning scheme will therefore still be able to infer useful information.

While the stability condition is therefore not overly constraining, if it does not

hold, the execution time for the next invocation of C within the application may not be sampled from the distribution observed over the prior few invocations of C. In that case, the choice of n will be less optimal with regards to maximizing expected speedup and minimizing resource waste.

There is often good reason why the stability assumption holds for a variety of applications. The characteristics of the input data significantly determine the execution-time of C. For example, in TSP, the size of the graph and the degree of nodes significantly determine the execution time. A given invocation of the application is likely to invoke C using data whose characteristics fall within a relatively narrow range. Therefore, the behavior of C on the next input will be drawn from the distribution of behaviors on previous inputs which will validate the stability condition.

For example, an application planning routes for delivery trucks would repeatedly invoke TSP over the same graph, but with perhaps different constraints for each truck. In all of our sample applications, the stability condition held.

2.3.2.2 Formal problem

The general problem is, given a set S of algorithms to solve a N-way computation C, which combination of ways taken from S has the best parallel efficiency. Note that there are two separate components to this: **i)** which algorithms to pick and **ii)** for algorithms that have inherent diversity (such as randomized algorithms) how many instances of that algorithm to run. In the following sections, two orthogonal special cases are studied: **i)** C is solved by a randomized algorithm R and **ii)** C is solved by a set of non randomized heuristics (or parametrized algorithms) H_1 to H_q.

2.3.2.3 Randomized algorithms

In a randomized algorithm R, a random seed determines the behavior of the algorithm, and in particular its execution time. For a given input I, the execution times of the algorithm will be distributed according to a probability distribution function (PDF)

and while the exact execution time of a run on I is unknown, an *expected* completion time can be computed from the PDF.

For a specific input I, a PDF can be learned by repeatedly invoking the algorithm on I but it will not be the same across inputs. However, under the stability assumption, the characteristics of the inputs vary slowly; in other words, it can be assumed that:

$$\text{PDF}\left(I_{j+1}\right) \approx \text{PDF}\left(I_1 \ldots I_j\right)$$

Therefore, R's PDF for previous inputs is assumed to apply to the next input.

The goal is to pick n that maximizes $\text{P}_{\text{Eff}}\left(n\right) = S_n/n$. To do this, for each $n = 1, 2, \ldots$, the runtime estimates S_n using the estimated PDF. Let $F_1\left(n\right)$ denote the cumulative distribution function (CDF) corresponding to the estimated PDF:

$$F_1\left(t\right) = \text{Prob}\left\{\text{Execution time of } P < t\right\}$$

However, the CDF for the fastest completing way when n independent ways are launched in parallel is what matters:

$$F_n\left(t\right) = \text{Prob}\{\text{Execution time of fastest way } < t,$$
$$\text{when } n \text{ independent ways are launched}\}$$

It can be shown that:

$$F_n\left(t\right) = 1 - \left(1 - F_1\left(t\right)\right)^n$$

This is because the probability for each independent way to **not** complete within time t is $1 - F_1\left(t\right)$, and therefore the time for **none** of the n ways to complete in time t is $\left(1 - F_1\left(t\right)\right)^n$. The expected completion time E_n is then the *mean* over F_n: $F_n\left(E_n\right) = 0.5$. Now, $S_n = E_1/E_n$.

In practice In practice, the CDFs are maintained as step functions. Suppose a single instance of R is repeatedly launched and times t_1, t_2 and t_3 with $t_1 < t_2 < t_3$ are

observed, the approximation of F_1 will be updated with those three points and F_1 will be such that $\forall t < t_1, F_1(t) = 0$, $\forall t \in [t_1, t_2), F_1(t) = 1/3$, $\forall t \in [t_2, t_3), F_1(t) = 2/3$ and $\forall t \geq t_3, F_1(t) = 1$. Although crude, this allows a quick approximation of the CDF. When n instances of R are launched, F_n is updated instead of F_1 and since all CDFs can be derived from one another, they can all be simultaneously updated.

To adapt to the slow variations in input characteristics, each input point has a weight, decreasing the importance of older points.

With the CDFs, E_n can be computed for each n from which S_n can be deduced.

Selecting the optimal number of ways To select the numbers of ways to launch, S_1, S_2, etc. are computed and the runtime determines if the improvement in speedup is worth the extra resource cost. The specific criteria can be set by the programmer. This approach is greedy in the sense that if S_i does not meet the criteria S_{i+1} does not either and there is no need to compute it.

2.3.2.4 Distinct algorithms

In the case of distinct heuristics, randomness is no longer a source of uncertainty. However, uncertainty about the exact input that will be passed to the computation C remains. Since the inputs passed to C vary, it may not be possible to predict which heuristic will do better on the next input. However, the assumption that past input characteristics are representative of upcoming inputs (stability assumption) still allows the runtime to learn statistically meaningful information.

Mathematical model Given q heuristics H_1 to H_q, suppose each heuristic is run to completion on each input I_l to I_m. For each input point, the completion time of each heuristic is recorded:

$$e_{ij} = H_i(I_j)$$

The average completion time $E[H_i]$ of H_i is given by:

$$E[H_i] = 1/m \sum_{j=1}^{m} e_{ij}$$

Given the stability assumption on inputs, it can be assumed that the expected completion time of a given heuristic on the next input is the average time over past inputs:

$$H_i(I_{m+1}) \approx E[H_i]$$

This allows the estimation of the expected completion time of each individual heuristic for the next input I_{m+1}. If only one heuristic has to be picked to run (i.e., $n = 1$), the best choice is to pick the H_i with the smallest $E[H_i]$. However, the runtime needs to determine whether running multiple heuristics provides significantly greater speedup. For this, an optimal n and the set of n heuristics to launch in parallel need to be determined.

Consider a naive approach based on sorting by smallest $E[H_i]$ as follows: $\{H_{i_1}, H_{i_2}, H_{i_3}, \ldots\}$ where for $n = K$ $H_{i_1}, \ldots, H_{i_K}$ get launched. However, as Figure 4 shows, this approach is suboptimal. The figure shows execution times for heuristics on inputs I_l to I_m, with the sorted heuristics $\{H_2, H_3, H_1\}$. For $n = 2$, launching $\{H_2, H_3\}$ as per the naive approach performs worse than launching $\{H_2, H_1\}$, based on fastest completion times. Even though $E[H_3] < E[H_1]$, H_3 is always slower than either H_1 or H_2, hence selecting it is not useful (even for $n = 3$).

The above illustrated a need for a general case test (for $n > 1$) that takes input-dependent orderings of heuristics into account. Therefore, in selecting the best H_{i_k} in addition to H_{i_1}, the following expected completion time needs to be minimized:

$$E[H_{i_1}, H_{i_k}] = 1/m \sum_{j=1}^{m} \min(e_{i_1 j}, e_{i_k j})$$

Further, $n = 1$ versus $n = 2$ can be decided based on:

$$E[H_{i_1}, H_{i_k}]/2 < E[H_{i_1}]/1$$

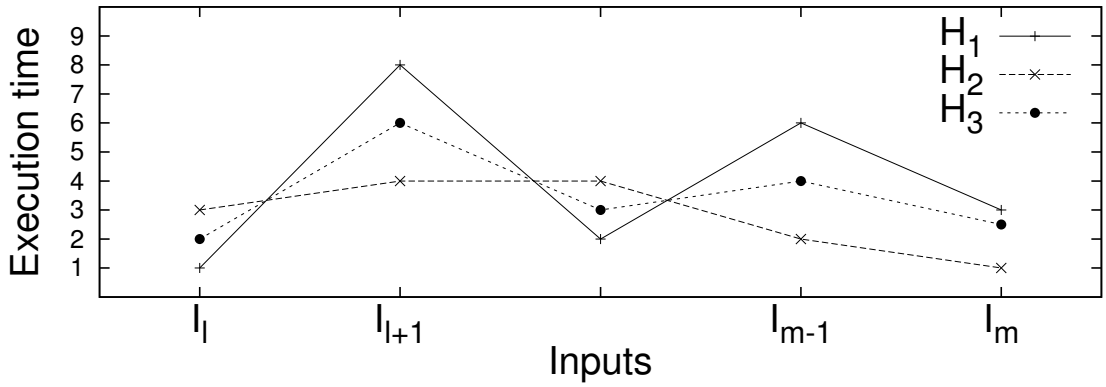


Figure 4: Execution times for heuristics H_1, H_2, H_3 on inputs I_l to I_m

where the best H_{i_k} is picked. This inductively generalizes to answering $n = 3, 4, \ldots$ and picking the best H_i to add at each step. For $n = 3$ for example, pick the best H_{i_q} minimizing:

$$E\left[H_{i_1}, H_{i_k}, H_{i_q}\right] = 1/m \sum_{j=1}^{m} \min\left(e_{i_1 j}, e_{i_k j}, e_{i_q j}\right)$$

and also ensure that:

$$E\left[H_{i_1}, H_{i_k}, H_{i_q}\right]/3 < E\left[H_{i_1}, H_{i_k}\right]/2$$

Note that the above strategy is not optimal since it greedily add only one additional heuristic at each step. Ideally it should test every possible subset of size 2 against H_{i_1}, and then every possible subset of size 3 against the best subset of size 2, and so on. Such an approach has an exponential complexity, making it unsuitable in practice.

In practice The theoretical model assumed measurement of the completion time for each heuristic on each input. In practice, this is obviously not wanted as it negates the benefits of N-way by forcing the runtime to wait for all heuristics to complete. Therefore, the runtime will only wait for the *first* heuristic to complete and record its completion time T. For all other heuristics, the runtime *approximate* their completion time on the input in the following way:

- The completion time of heuristics that were run but did not complete is cT where $c > 1$ is a parameter defined in the runtime. Indeed, the completion time of these heuristics is greater than T and the programmer controlled c parameter indicates how strongly winning heuristics should be favored.

- The completion time of heuristics that were *not* run (for example, if they were not picked) is dA where A is the average completion time for that heuristic over past inputs and $d < 1$ is a programmer controlled parameter controlling how often untried or cast-away heuristics should be given a chance. This allows the learning algorithm to adapt to changing conditions by ensuring that ways that were previously discarded are periodically evaluated.

The two heuristics above allow the runtime to estimate the completion time for all heuristics on all inputs and can then apply the mathematical model described in the previous section. Similar to the randomized case, the learning model also "forget" older points to allow adaptation, restricting to a window I_l to I_m of prior invocations of the computation C.

2.3.2.5 Generalized algorithm

The models described are efficient learning models for the special cases of randomized and heuristics-driven C. A more generalized model combining both for the same C is possible (TSP can combine randomization and heuristics), but such a model is the subject of future work given its high computational complexity in its current form.

2.3.2.6 Resilience of the learning models

The N-way model was designed to make use of the idle cores and therefore assumes that each N-way thread is given exclusive access to a core. However, even if transient loads occur that may perturb this assumption, the learning models described are resilient. This is due to the fact that the effects of even a relatively large perturbation

will only impact a few samples and be mitigated by the other samples. A sufficiently large and persistent perturbation however will clearly compromise these models but only for a short duration after it goes away as the models use only recent samples (using a sliding window).

Furthermore, even if perturbations do affect the estimations of expected completion times, the decision process of the N-way model will not always be compromised. Consider the choice between n ways versus $n+1$ ways. Even if the estimations of E_n and E_{n+1} are perturbed, the N-way model will make the same decision as long as the ordering of $\mathrm{P_{Eff}}(n)$ and $\mathrm{P_{Eff}}(n+1)$ does not change. Therefore, the model can tolerate significant errors in the estimations of the expected completion times as long as the resulting parallel efficiencies compare the same way.

2.3.3 Support for Quality-of-Result

The above models have been described in the context of providing speedup. In that case the measurable quantity is execution time and is lower bounded by 0 which represents the best possible case (ie: an algorithm that takes no time to execute is the best case). The only assumption that the learning algorithm makes is that "smaller is better". It can therefore be extended to provide QoR improvements if the programmer can provide a measure for quality at the end of the way and a "distance" that quantitatively compares two qualities. The runtime can then learn the distribution of qualities instead of execution times.

2.4 Efficiency through culling

The learning framework proposed in Section 2.3.2 tries to a-priori pick n to maximize the parallel efficiency P_{eff}. However, as the various ways execute, more information may be gained about their progress which can enable the dynamic *culling* of ways which are almost certainly *not* going to "win". This will reduce the overall utilization of processor cores (to n_{eff}) while having very little impact on the expected speedup

(as only ways that will "lose" will be preemptively culled). This section proposes a culling framework. The motivation is to **i)** relieve system pressure on shared resources in particular and **ii)** improve energy efficiency.

2.4.1 Notion of progress

To determine the ways to cull, the runtime relies on a programmer-supplied notion of *algorithmic progress*; in other words, how far has the algorithm come in solving the computation. This notion is not always easy to characterize and is very computation specific but is relevant in a number of cases:

- Computations that are solved using **greedy constructive** algorithms have a natural notion of progress as both the amount of work done and to-do are readily available.

- Other computations, such as optimization problems, are solved by finding partial solutions which are subsets of the final solution: dynamic programming, incremental algorithms (Dijkstra's single source/destination), etc. The 'size' of the partial solution is a good measure for progress.

To indicate progress, the programmer must define a normalized metric M that takes values between 0 and 1 where 1 indicates completion[2].

2.4.2 Culling mechanism

While the progress metric M provided and updated by the programmer gives absolute progress for each way, *progress per resource* is actually what the runtime wants to compute because it wants to eliminate the ways making the least progress but consuming the most resources. However, the N-way system only currently considers the CPU as a resource and M^s will therefore denote the measure of progress per CPU-time As part of future work, we are looking at measuring other resources consumed

[2]Note that M may be derived from auxiliary metrics if this is easier to express for the programmer

by each way (such as L2 footprint, bus activity, etc.) through the use of performance counters.

Assumptions Metric monitoring provides the runtime with information about past progress. The runtime therefore assumes *linearity* in progress to extrapolate future progress. This simplifying assumption is constraining but the programmer can update M to best approximate this assumption and the culling system can tolerate this approximation.

Culling relies on answers to two questions: **i)** *when* to look for ways to cull and **ii)** *what* ways to cull.

When to cull? Ideally, the runtime would like to cull inefficient ways as early as possible but it does not want to constantly monitor progress to limit overheads. Therefore, it attempts to check only when it is *likely* that ways can be culled.

In the current mechanism, each way W reports its progress M_W to the runtime which then decides whether or not to attempt culling. It will be attempted if either **i)** a way has made significant progress, **ii)** sufficient time has elapsed, or **iii)** a sufficient number of progress reports have occurred since the last culling attempt. The runtime has access to the last reported progress information from each way.

What to cull? Given the linearity assumption, progress metrics M^s_W can be directly compared. The goal of the culling algorithm is to quickly determine *all* underperforming ways that have little chance of "catching-up" with the best performing ways.

To do so, the algorithm clusters all ways based on their M^s. The idea is to group together ways that are making a similar amount of progress. The number of clusters is not fixed a-priori instead relying on a hierarchical bottom-up clustering approach which stops when the mean squared error (MSE) of the merged cluster D of A and

B is greater than the weighted sum of the MSEs of A and B where:

$$\mathrm{MSE}(D) = \sum_{W_i \in D} \left(M^s_{W_i} - M^s(D)\right)^2 / |D|$$

We will cull the worst cluster D_{worst} if

$$\alpha\left(1 - M^s(D_{worst})\right) > 1 - M^s(D_{best})$$

where α is smaller than 1 (currently 10%) and M^s of a cluster is the mean metric of ways in that cluster. This ensures that culling is less aggressive when the best ways have made little progress (and it is still unclear who will "win") and more aggressive when the best ways are closer to finishing. This also allows the linearity assumption to be relaxed and progress need only be mostly linear.

2.4.3 Compatibility with learning

Culling is fully compatible with the learning N-way model under the assumption that culling *only culls ways that would not have "won"*. This is a reasonable assumption as the culling algorithm kills off ways that are making little progress and are therefore highly unlikely to "win". Under this assumption, culling has no impact on learning as it does not modify the information used by the learning algorithm, namely the completion time of the winning way.

Culling is optional and does not directly provide additional speedup. However, a speedup can be achieved indirectly by reducing contention for shared resources. Even if the programmer cannot provide progress monitors, the learning approach is still fully applicable.

2.5 Implementation

The N-way model is supported by a runtime to enable dynamic monitoring and adapt the selection of ways as the characteristics of the inputs changes. The runtime has the following key roles: **i)** determine the `Types` of algorithm to launch and how many

of each `Type` (in the case of a randomized algorithm), **ii)** optionally monitor progress and cull any under-performing way and **iii)** provide isolation among the ways and keep them side-effect free.

2.5.1 API

All code examples provided here are in pseudo C-like code. The actual implementation makes heavy use of C++'s templating abilities to provide stronger type-checking guarantees.

The N-way model requires very little from the programmer who needs to identify:

- A diverse computation C and the different N-way kernels available to solve it.
- An optimization objective: speedup or QoR improvements. In the case of QoR improvements, a quality metric must be provided.
- To support culling, an optional specification of progress for the ways is required.

The code example in Figure 5 will be used to illustrate the API.

Specifying a N-way computation Specifying a N-way computation is simply a matter of specifying the various algorithms (N-way kernels) that are possible. This is shown on Lines 10 to 12 where three different function pointers are attached to the same N-way computation `findAPath`. Heuristics, parametrized algorithms and randomized algorithms which can be launched multiple times can all be added. The programmer is responsible for identifying the functions that "belong" to the same N-way computation.

Specifying an optimization-objective The programmer must specify whether he wishes to obtain speedup or QoR through an *optimization objective* as shown on Lines 15 and 21. Note that in the common case of speedup, the programmer

```
struct input_t {
  Node start, end;
} myInput;

struct findAPath_t {
  int pathLength; bool pathFound;
} findAPath;

/* heur1_f, param_f and rand_f are function pointers */
findAPath.addAlgo(heur1_f);
findAPath.addAlgo(param_f, parameterSpace);
findAPath.addAlgo(rand_f, nonEnumerable);

/* Two example switches */
struct chooseFastest_t {
  StopWhen = (0, 0);
  MinimumRequired(findAPath_t f)
  { return f.pathFound; }
} chooseFastest;

struct chooseShortest_t {
  StopWhen = (undef, 1);
  MinimumRequired(findAPath_t f) { return f.pathFound; }
  Comparator(findAPath_t f1, findAPath_t f2)
  { return f1.pathLength - f2.pathLength; }
} chooseShortest;

findAPath.run(myInput, chooseFastest);
findAPath.run(myInput, chooseShortest);
```

Figure 5: N-way pseudo-code for a path-finding problem.

has nothing to specify as a default 'speedup objective' is provided. The optimization objective becomes relevant when QoR is traded off with execution time. The optimization-objective determines *when* and *how* the runtime chooses the winning way.

We provide the following APIs:

- `StopWhen` is a tuple (p_t, p_c) where $p_t \geq 0$ and $0 \leq p_c \leq 1$. The idea behind both numbers is to allow a trade-off between the time that the runtime waits and the number of ways it waits for. Indeed, to obtain maximum QoR, the programmer

would ideally like to wait for all ways to complete to pick the absolute best but in practice, he might want to cap the runtime of the N-way problem thereby trading off quality with execution time. If t_1 is the completion time of the first way, and $n_c(t)$ is the number of ways that have completed by time t, t_d the time the runtime will make a decision is defined as the minimum t such that either $t_d \geq (1 + p_t) * t_1$ or $\frac{n_c(t_d)}{n} \geq p_c$ is true. Two such specifications are shown in Lines 16 (maximum speedup) and 22 (maximum quality).

- `Comparator` qualitatively compares two ways to determine which provides the best QoR. The computation is based on the metrics defined for the N-way computation (Line 6) which the programmer must update before the way completes.

- `MinimumRequired` is also a comparator that determines if a way completed with a valid result. Indeed, it is frequent for heuristics to return an answer of the type "I have not found a solution". While semantically valid, the programmer may wish to return a solution if at all possible. The programmer can specify what condition a "good" solution must meet. Solutions not meeting this requirement are considered to still be computing (and therefore take infinite time). Note that if no "good" way is found, the first way to return will "win".

Again, these functions are only required in the case of QoR objectives and allow great flexibility in specifying a meaningful quality objective.

Launching a N-way computation Lines 28 and 29 both show an invocation of a N-way computation: the first one will run the N-way computation and pick the first way to finish while the second will pick the one returning the shortest path. The programmer would replace the execution of the N-way computation B (in Figure 3(a)) by one of these two calls to obtain its N-way execution. During N-way execution, one and only one way executes on each selected core. In other words, if the runtime

determines that only one core should be used, only one way (picked by the runtime) will be launched. For n cores, n ways will be launched, one per core.

2.5.2 Progress monitors

The optional culling relies on ways reporting on their progress. In this implementation, each way is passed an individual copy of a set of `Metrics` as shown on Line 6 (the `Metrics` are, in this case, the two variables ‘`pathLength`’ and ‘`pathFound`’. Each way maintains a copy of these computation-specific values that the N-way runtime can read from each of the ways' states whenever required. Note that for efficiency reasons, the `Metrics` are implemented as a double buffered data structure which enables the runtime to read a consistent `Metrics` state while not blocking or otherwise impacting any of the ways. The programmer must also provides a function that takes as input the `Metrics` and returns a single progress value (M_W^s).

2.5.3 Providing isolation

A very important aspect of the API and runtime is to enforce isolation between ways. The goal is to encapsulate each way so that when destroyed, no trace of its activity remains.

Since each individual way is implemented as a thread rather than as a separate process the wrapping of non-local memory accesses to provide isolation among the threads is required. The choice of threads versus other mechanisms such as “fork” on Linux is justified in Section 2.5.4.1.

Currently, the implementation does not handle certain system calls (such as writes to a file or device) but these can be wrapped (or intercepted using ‘ptrace’ for example on Linux) as well if required. The current framework handles two aspects of isolation:

- Garbage collection of all heap-allocated memory in discarded ways.
- Wrapping of *non-local state* which is defined as anything visible outside the

scope of a way (globals or heap variables are accessible from outside the scope of the way).

Garbage collection is implemented in a distributed manner: each thread is responsible for its own "garbage" which it clears before being terminated.

2.5.3.1 Wrapping non-local state: a lightweight versioning system

The runtime relies on a lightweight versioning of data where each non-local variable is implemented as a vector of pointers, one for each of the executing ways plus a default pointer representing the latest "official" value. Each way-pointer points to a lazily created private copy of the variable. Figure 6 shows a variable `Foo` in a situation where 4 ways are running (on threads T_1 through T_4). Ways that do not have a private copy will use the default copy in read-only mode (labeled D in Figure 6). Three operations on the variables are defined:

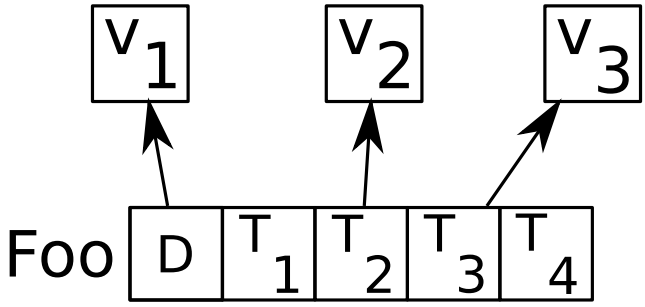


Figure 6: A non-local variable `Foo` in the isolation system

- **Read**: The private value for the thread, or default value if none exists, is returned. The value returned cannot be modified (`const` in C++). This is an $O(1)$ operation.

- **Write**: A write will trigger an automatic copy-on-write if this is the first write access. Subsequent writes will modify the private value. The COW approach minimizes memory overhead as well as copy overhead. This is an $O(1)$ operation as well (minus the copy itself).

- **Commit**: This operation serves to make public a way's private copies and is implemented as a simple pointer change (no copy). The private copies of the other ways as well as the old default value are destroyed. Destruction is $O(n)$ in the number of ways for each variable and can be distributed across the unpicked ways.

Figure 6 shows the state of `Foo` after T_2 and T_3 have performed read access to and and T_1 and T_4 have either only read it (and would therefore read the value in V_1) or not accessed it at all.

To minimally modify the program, C++'s templating capabilities is used through a type, `NVShared<T>`, which behaves exactly like `T` except that accesses go through the isolation system. For example, in Figure 6, `Foo` would be declared as `NVShared<T> Foo` where `T` is the C++ type of `Foo`. Access to `Foo` from thread T_2 would be equivalent to accessing V_2 while access from T_1 would equate accessing V_1 if read-only or a newly created copy of V_1 otherwise. The API also defines a `NVPtrToShared<T>` to wrap addresses to `NVShared<T>`. For example, `NVPtrToShared<T> AddrOfFoo` would return the address of V_2 if accessed from T_2. The only requirement of the system is that `T` provide a deep-copy mechanism to be able to correctly duplicate objects when needed.

2.5.4 Thread-based implementation

The ways are run using a pool of worker threads. At the start of a N-way program, the runtime launches a fixed number of worker threads which will wait until a specific way is assigned to them. A modified pool based on Boost threads is used but with added support to cancel a running way without also canceling the underlying worker thread. This allows the quick reuse of the worker threads without the need to respawn them.

2.5.4.1 Comparison with a fork mechanism

On UNIX based systems, the "fork" mechanism could also be used as it would naturally provide an OS supported isolated address space and also be efficient given its copy-on-write implementation [6]. The main advantage of the fork mechanism is that the original code could run unmodified although the Clang pass mitigates this. However, the fork mechanism also has downsides:

- The cost of a fork is much higher than that of assigning a way to a thread in a pool (mechanism presented here).
- The fork mechanism is not implemented in every OS. Windows for example does not have an equivalent mechanism. The presented framework only relies on the ability to create threads which makes it much more OS independent.
- The isolation mechanism used provides a natural way for the runtime to monitor the different ways. This aspect of the runtime was detailed in Section 2.4.
- The isolation mechanism provided also allows a much more fine-grained control of the data that will be copied. Indeed, the fork mechanism's COW will only function at the granularity of a page, which may not be optimal if the data written by each of the ways is very small or very scattered. The mechanism provided here minimizes the amount of data that has to be copied.

Both mechanisms therefore have their advantages. The validity of the N-way model is independent of the mechanism used to provide isolation and a different implementation would provide similar benefits.

2.5.5 Debuggability

The fact that the programmer does not know a-priori which of the ways will commit may seem like a debugging nightmare. However, since the ways are well encapsulated, debuggability will not be much harder than for sequential programs and

definitely simpler than for traditional parallel programs. If each way is a sequential kernel, debugging each one of them individually in the context of the entire program without N-way parallelism is sufficient to ensure that the resulting N-way program is debugged. None of the hard to debug parallel bugs like deadlock, live-locks and races are introduced. Repeatability of execution can also be obtained by recording the choices made at each commit point.

2.5.6 Automated compiler transformation of a program for N-way

To lessen the burden on the programmer, a compiler based approach relying on the C++ frontend Clang [15] was developed which allows source-level analysis to identify non-local variables. A source-to-source translation from the original code to a N-way code where non-local variable definitions get wrapped with $NVShared$ can therefore be performed. The programmer only needs to identify the functions to be converted to N-way problems.

The transformation of a regular sequential program to one that supports N-way consists of:

- The programmer identifies the various functions that "belong" to the same N-way problem.
- Optionally the programmer identifies `Metrics` for the N-way problem to support culling and/or a QoR objective. He modifies appropriate functions to updated these `Metrics`.
- Running the source code through the Clang based translator.
- Compiling the program normally. The runtime is implemented as a shared library which should be linked against the program.

The burden on the programmer rests principally in identifying and expressing diversity in parts of the program. The Clang translator takes care of the source-level

transformations needed to correctly wrap non-local variables and the runtime takes care of the complex process of determining the exact ways to run to maximally benefit from the N-way model. If no benefit can be extracted, the N-way system will approximate a sequential execution.

2.6 Experimental results

The experiments seek to validate the following:

- Brute-force N-way parallelism provides speedup and QoR improvement.
- The learning approach allows the N-way model to provide significant speedup while requiring significantly fewer resources. In other words, the effective utilization of the machine n_{eff} is substantially reduced by the learning model.
- Culling further reduces n_{eff} with little impact on the expected speedup obtained via the N-way model.

The experiments also demonstrate the low-overhead and scalability of the runtime.

All tests were run on a 64bit Linux Ubuntu System running a dual quad-core Xeon E5540 at 2.53 GHz with 12GB or RAM. GCC 4.4.3 was used to compile the roughly 10000 lines of runtime code with "-O3".

2.6.1 Benchmarks

N-way was applied to the following benchmarks to demonstrate the above points.

- WalkSAT is a randomized SAT solver that has shown good performance in SAT competitions. At each step, the solver picks a random unsatisfied clause and uses a heuristic to select the variable in the clause to "flip" to satisfy the clause. The SAT inputs are representative inputs taken from DIMACS [24]. To keep running times reasonable, the program was terminated after 5 minutes if no solution was found. This randomized benchmark will show that N-way can

extract speedup from a purely sequential algorithm. Note that we do not seek to demonstrate the fastest SAT solver available but simply that n-way has a potential in combinatorial problems like SAT solvers.

- The MSL library is a motion planning library that uses rapidly-exploring random trees (RRTs) [45] which are designed to efficiently search a high-dimensional non-convex space. The library was used in a hand crafted simulation to demonstrate the limits of the N-way learning model as well as the impressive super-linear speedups that can be obtained.

- TSP-solve is a TSP solver implemented by Chat Hurwitz which contains a variety of heuristics.

- GALib [82] is a library providing support for genetic algorithms (GAs) which was used to again solve the TSP. Genetic algorithms rely on randomness to create successive "generations" of solutions. This benchmark demonstrates how N-way can be used to improve the quality of the final generation.

- ListSched: A simple greedy list-scheduler using three different heuristics to pick the node that gets scheduled next (if more than one is available). This benchmark, although modest in size, demonstrates the usefulness of culling as well as QoR improvements with a very small amount of diversity.

For some of these benchmarks, specialized traditional parallel implementations exist; however, the aim is to show the potential of an orthogonal parallelization strategy that relies on diversity. The fact that an orthogonal form of parallelism is also possible does not detract from the value of diversity-based parallelism (N-way). The applicability of both approaches will depend on the characteristics of the application: sometimes a traditional approach will be more natural (when divisibility of task/data is naturally present) and sometimes a N-way approach will be simpler (for irregular algorithms for example when it is difficult to determine dynamic dependencies).

The baseline for all speedup results is the unmodified sequential code (without any N-way modification)

2.6.2 Speedup through randomness

In the graphs, speedup is relative to the original unmodified C++ code in which only a single instance of the randomize algorithm is run. The slowdown shown for the '1-thread' case therefore represents the overhead of the system. Speedup is shown for a 'brute force' approach where the N-way system is forced to use a fixed number of ways as well as for the learning and the learning+culling approach (when appropriate). In those cases, the number of threads corresponds to the *maximum* number allowed to the runtime. All results are averaged over multiple runs.

2.6.2.1 WalkSAT

Figure 7 gives WalkSAT results. For very small inputs (`f400`) a slowdown occurs due

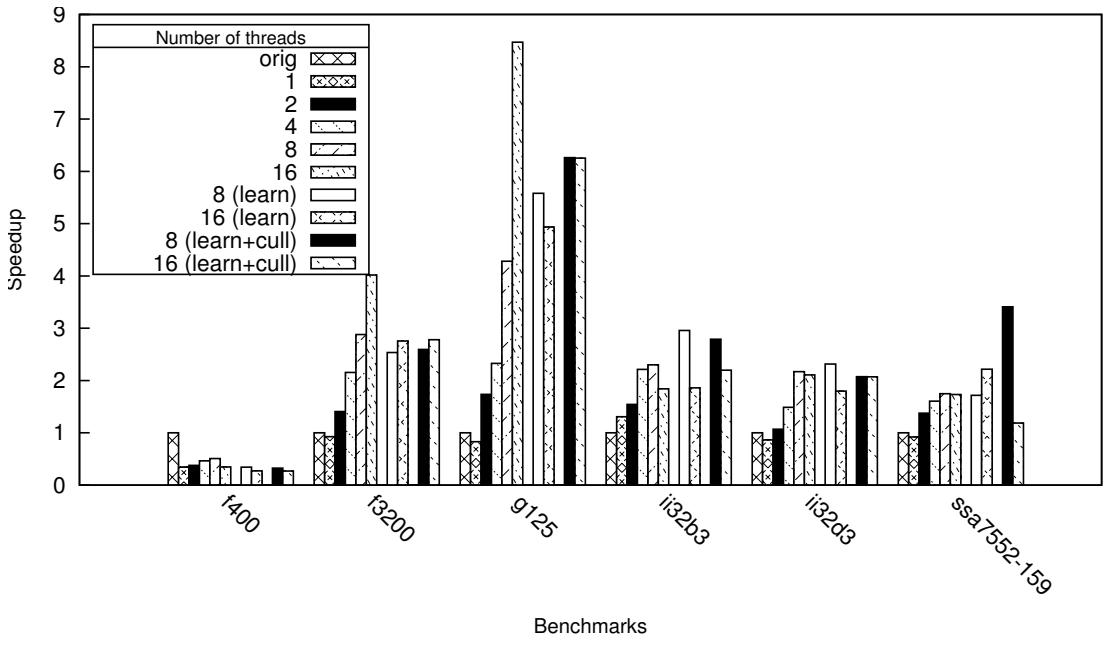


Figure 7: N-way speedup results for the WalkSAT benchmark

to the overheads of the N-way system but overheads are negligible for larger inputs. For larger inputs, the benefit of N-way becomes clear as significant speedups obtained

in particular for many threads.

Speedup a function of diversity The speedup obtained depends on the variation in execution time of the original program for a given input. In general, a broad correlation exists between the speedup obtained through N-way and the Coefficient of Variance [3] (CoV) which is representative of the amount of diversity present: the larger the CoV, the more speedup potential there is. For example, the CoV for `ssa7552-158` is 0.73 compared to 0.82 for `f3200`.

2.6.2.2 MSL motion planning

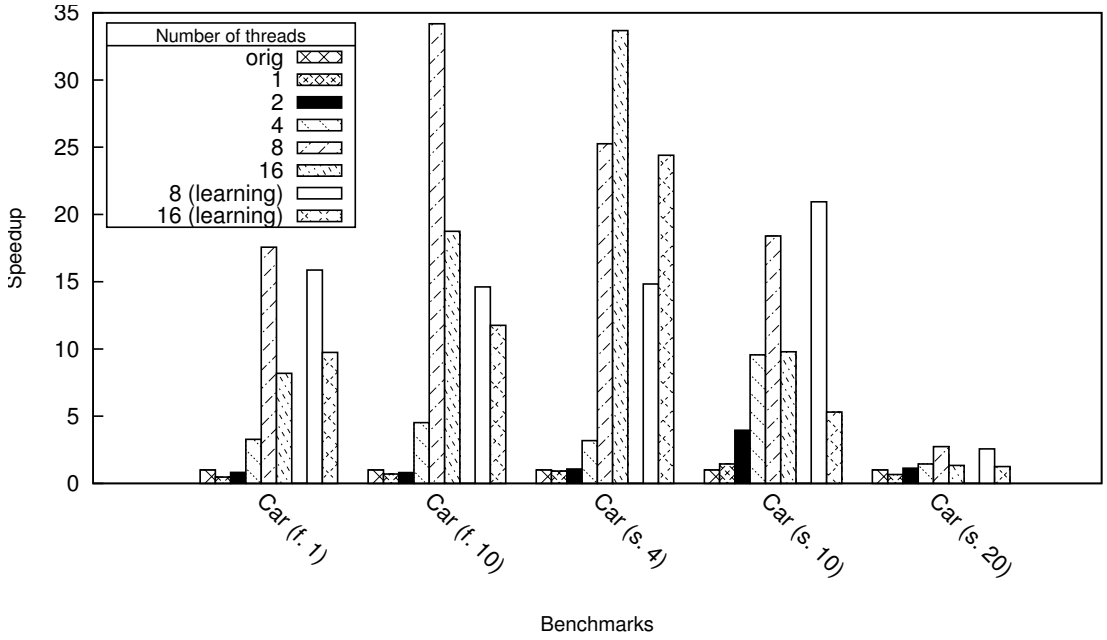


Figure 8: N-way results for the MSL benchmark. The benchmark names represent either a fixed opening size (`s.`) or a periodic increment in the opening's size (`f.`).

Results for the hand-crafted `Car` benchmark are shown in Figure 8. The benchmark simulates a car that has to find its way through a narrow opening and showcases the strengths of the N-way framework as the benchmark is similar to "finding a needle in a haystack". The random algorithm may find it very quickly (short execution

[3] defined as the ratio of the standard deviation by the mean

time) or may take a very long time. The CoV is thus very high (1.33 for an opening of size 4) and super-linear speedups are therefore possible. The benefits tapper down with larger openings as the probability of quickly finding the opening increases.

2.6.2.3 Effects of learning and culling

In Figure 7, bars 7 and 8 show the speedup obtained when using the learning approach that tries to launch "just enough ways" to obtain a good speedup for each input. The runtime is given a *maximum* number of cores to utilize (8 for bar 7 and 16 for bar 8) and may choose to launch anywhere from 1 to that maximum number of ways (at most one per core). Therefore, across all inputs, the effective utilization of cores, n_{eff}, is lower (the runtime will not always choose to launch 8 or 16 ways).

The last two bars (9 and 10) show when the learning approach is compounded with the culling approach. In this case n_{eff} is further lowered as certain ways will be culled before the N-way computation completes. In both cases, the parallel efficiency is defined as $\mathrm{P_{Eff}} = \frac{S}{n_{eff}}$.

Table 1: Maximal N-way speedup S obtained and the corresponding parallel efficiency for the fixed case for WalkSAT. The fixed n this was achieved for is shown in parentheses. The same quantities are shown when learning and learning+culling is applied for 8 threads

Dataset	Fixed		Learning		Learning and Culling	
	S	E	S	E	S	E
f3200.cnf	2.88 (8)	.37	2.53	.34	2.59	.57
g125.17.cnf	4.28 (8)	.56	5.58	.79	6.26	.89
ii32b3.cnf	2.30 (8)	.31	2.95	.42	2.79	.39
ii32d3.cnf	2.16 (8)	.29	2.31	.32	2.07	.28
ssa7552-159.cnf	1.74 (8)	.24	1.71	.24	3.41	.47

WalkSAT benchmark Figure 9 illustrates the utility of learning. One can see that the speedup obtained with N-way by increasing the number of cores tappers off after 8 cores. This means that the additional cores are not being effectively used by N-way (the diversity present in the algorithm is sufficiently expressed with fewer than

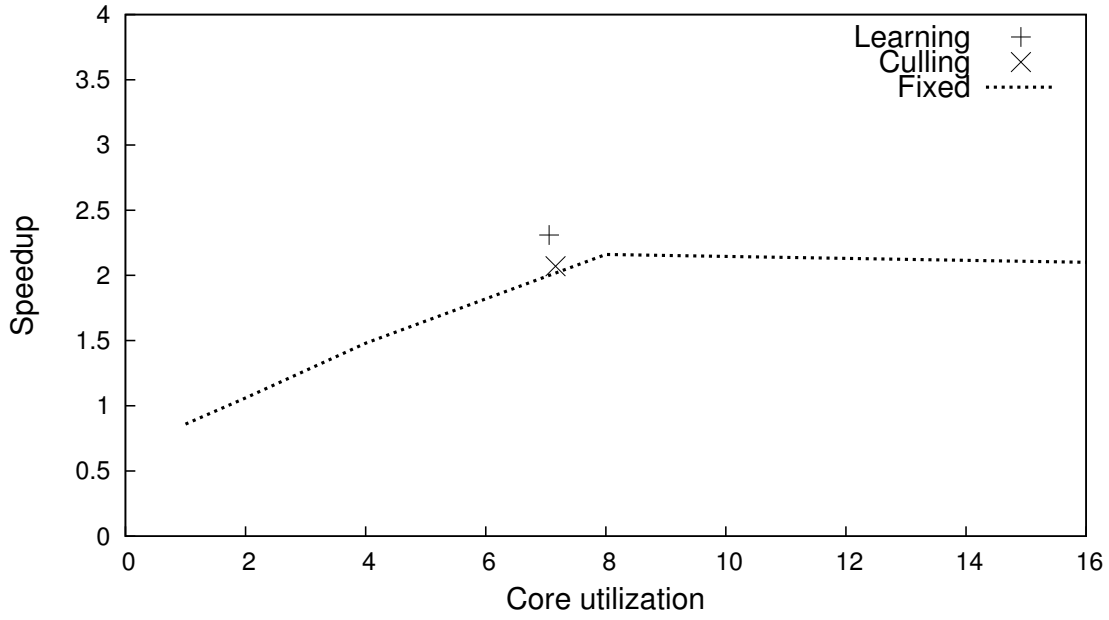


Figure 9: N-way speedup versus core utilization for the ii32b3 input to WalkSAT. The line shows the evolution of the speedup if a fixed number of cores is used (without learning or culling) and the points.

8 cores). One can see that the runtime correctly chooses the run the benchmark with $n_{eff} = 6.97$ obtaining a good speedup while not committing unnecessary cores. For the other benchmarks, Table 1 summarizes results for all WalkSAT inputs.

It is important to note that the learning scheme is designed to optimize for parallel efficiency (P_{Eff}) and not for maximal speedup. If the learning scheme was designed for maximal speedup, it would always choose to launch as many ways as possible. The scheme, however, does not try to maximize speedup and therefore the speedups obtained by the learning scheme can be lower than those obtained for the basic N-way scheme that does not integrate learning and culling: the speedup may be higher utilizing more ways but the efficiency is worse as shown in Table 1.

In most cases, efficiency increases when learning is used and increases even more when culling is also used. This indicates that, on average, fewer cores were used to obtain a similar speedup. For `f3200`, learning actually lowers efficiency; this is due to the fact that `f3200` is still relatively small and uses few iterations which limits

the possibility to learn. As expected, the addition of culling increases efficiency while providing very little speedup gains. Any gains are most likely due to the freeing up of shared resources (such as the L2 cache) by the culled ways. This is particularly visible in the large `ssa7552-159` dataset. Note that although it is hard to provide a good progress measure for a SAT solver, the addition of culling does not negatively impact performance and actually performs rather well even in these difficult conditions.

MSL For the MSL benchmark, the case of the `Car` benchmark with the size of the opening changing is interesting. One can see in Figure 8 that while the learning algorithm performs well for `Car (f. 1)` where the size of the opening increases slowly (by 1 every iteration), it performs very badly when it increases by 10 every time. This is caused by the underlying distribution of execution times changing too rapidly (since the inputs can rapidly go from a small opening size where diversity is useful to a large opening size where diversity is not). Therefore, the assumption that the inputs will be similar to one another or evolve slowly is broken. In this case, one can see that the learning model cannot perform as optimally as it could have but does not incur a slowdown.

Culling is not applicable to MSL as the notion of progress is hard to define.

2.6.3 Speedup through heuristics

Diversity through heuristics can also be used to obtain speedup: `tsp_solve`, a program containing different TSP heuristics, shows this. Note that no randomness was used in the heuristics.

In this experiment 50 randomly generated TSP problems were solved and the total execution time for all 50 problems was measured as well as the average tour length. The results are summarized in Table 2. When running a fixed number of threads, the heuristics are chosen at random while with learning, the runtime picks the best heuristic(s). Learning does not significantly lower speedup and reduces

Table 2: N-way results for a fixed set of 50 randomly generated TSPs.

Threads	Time (s)	Speedup	Avg. Tour Length
1	1191	1.0	8497
2	828	1.44	8463
4	335	3.56	8452
8	298	4.0	8445
16	295	4.04	8443
16 (learning)	300	3.97	8444

resource use: the learning approach effectively uses about 3 cores as opposed to the 16 used in the brute-force approach. Closely studying the results confirm that only two or three heuristics (out of a total of 13) frequently "win" with the others providing only marginal benefits. The runtime is able to correctly select those. Note that all heuristics are useful as, for each heuristic, there are inputs on which it performs better than the other heuristics. The input data set favored a few but, as a programmer, it is impossible to know a-priori if this is always the case. The N-way model can dynamically determine the useful heuristics for the programmer.

2.6.4 QoR through randomness

In GAs, the "fitness" of a population is a natural quality measurement. In the TSP example, a "fitter" individual is one representing a shorter path. Furthermore, the randomness in GAs has very little effect on the time each generation takes to be created but significantly impacts the fitness of the population. Therefore, GALib is a good candidate to demonstrate QoR improvements with N-way.

To evaluate the QoR impact of N-way, the algorithm was run for 2500 generations and compared the "fitness" of the best individual at the end of the 2500 generations. At each intermediate generation, the fittest generation was picked as the "winner" and it was fed to the next step. Results for various inputs from TSPLIB [68] are shown in Table 3. For learning and culling, effective resource utilization (in number of cores) is shown in parenthesis. N-way clearly improves the final population's fitness, thereby

Table 3: "Fitness" of the population after 2500 generations (smaller is better) as well as resource utilization for learning (marked 'l') and learning+culling (marked 'l+c').

Threads	kroB150	a280	lin318	rat575
Original	44.7	7.8	139	31.5
1	44.8	7.8	139	31.5
2	44.8	7	134.3	29.3
4	41.6	6.8	128.2	28.7
8	39.1	6.5	122.3	27.7
16	38.4	6.3	113.8	26.7
8 (l)	42.4 (4.9)	7.1 (4.6)	126.9 (4.5)	28.7 (4.1)
8 (l+c)	45.1 (3.5)	7.1 (3.5)	132.5 (3.9)	29.1 (3.8)
16 (l)	40.1 (8.3)	7 (7.0)	128.9 (7.3)	28.9 (6.6)
16 (l+c)	43.7 (8.1)	7.2 (7.0)	132.3 (7.2)	29.3 (6.6)

finding a shorter, or better path. More importantly, one also notes the benefits of both the learning and culling algorithms. For example, the difference in "fitness" between the case with 8 threads and 8 threads with learning is always less than 8%, thereby still giving the full benefits of the N-way model, yet the learning algorithm reduces n_{eff} by over 40% (average utilization of 4.5 for learning). Culling further slightly reduces quality but reduces resource usage by an extra 20%.

2.6.5 QoR through heuristics

Various heuristics to pick the next ready node to schedule were implemented in ListSched: pick the one with the longest critical path, biggest fan-out, etc. This benchmark is very small and runtime overheads dwarf possible speedups but N-way allows for QoR improvements. A set of directed acyclic graphs (DAG) to schedule was generated and each of the three original heuristics was run. N-way, which runs all three heuristics in parallel, was then run. In this scenario, since quality is the objective, N-way waits for at least 2 of them to complete. The best schedule is then selected. Waiting for only two limits the total execution time while still providing the benefit of quality improvement. The sum of the schedules found is listed in Table 4. Although the improvements are modest, they come for a low cost to the programmer.

Table 4: Sum of the schedules' lengths found over a set of DAGs for various heuristics

Heuristic or N-way	Sum
FanoutFirst	3129986
LongestDelayFirst	3133401
CriticalPathFirst	3131065
N-way (3 ways)	3115375

No single solution works as well as the N-way solution showing that a static choice is sub-optimal across all inputs. A domain expert could potentially add more heuristics and extract even better performance.

2.6.6 Runtime overhead and scalability

The speedup results in Figures 7 and 8 show that the absolute overhead of the system is low. Indeed, the basic overhead of the N-way system can be viewed as the difference between the first bar (the unmodified program) and the second bar ($n = 1$ N-way program) as the difference between those bars corresponds to the overhead of the N-way runtime and isolation system.

This overhead is further analyzed to pinpoint the "hotspots" in the runtime as well as its scalability. A home-grown high-precision profiler which outputs information identical to that of gprof except that it does so in an exact manner (no sampling) and only profiles marked functions was used. The results for a run of the WalkSAT benchmark on the `f1600` benchmark with 1, 8, and 16 threads are presented in Table 5. Non-shown runtime functions each represented less than 1% overhead. Other benchmarks present similar overhead characteristics. Not surprisingly, the vast majority of the overhead comes from dealing with access to non-local data. Close to 75% of the overhead is spent in accessing data and although the overhead of each call is relatively low, it does add up. While this overhead is non-negligible, it allows for the benefits described previously in Section 2.5.4.1.

The scalability of the runtime is also very good up to 8 threads: the fact that the time per-call stay mostly identical demonstrates this. However, the time per-call

Table 5: N-way runtime overheads for 1, 8 and 16 threads. "*Access (RO)*" corresponds to a RO access to a `NVShared`, "*thread info*" to a call to determine a running thread's characteristics and "*thread private*" to a TLS fetch.

Function	% total overhead			Call time (ns)		
	1	8	16	1	8	16
Access (RO)	53.78	48.49	49.72	12	13	26
Access (RW)	24.72	31.69	24.11	9	13	21
Thread info	14.95	13.92	18.53	2	2	5
Thread private	5.85	4.45	6.15	24	24	57

doubles when moving to 16 threads and this is most likely due to the dual-socket architecture of the machine. In the implementation, threads are bound to cores in increasing order and would therefore go off-chip for anything more than 8. This incurs additional overhead as the runtime must now communicate with threads located on two separate sockets.

2.7 Related work

There is strong algorithmic evidence of diversity in algorithms [53, 56, 39] and launching multiple instances of the same algorithm to speed up a computation has been used in some very specific instances: ManySAT [31, 84] for example uses a notion of "portfolios" that is very similar to the N-way approach. Other work in the security area [20, 70] seeks to exploit diversity to improve the security guarantees of a system. These works demonstrate the wide scope of diversity in algorithms (although they exploit it for a different purpose).

2.7.1 Competitive parallel execution

Trachsel et al. [77, 78, 76] introduced the notion of "brute force" N-way/CPE. The N-way model is very similar to this but introduces a statistical learning model and a culling approach that dramatically reduces resource consumption while preserving the speedup benefits. In addition to speedup, the N-way model also address maximizing QoR within the same framework. Both models were developed concurrently and

independently.

The N-way model allows the programmer to focus solely on expressing algorithmic diversity without having to worry about how to achieve speedup and at what cost: the runtime will dynamically ensure that speedup is maximized while resource usage is minimized. Coupled with the simple API, the N-way model presents a practical framework to exploit diversified computations to obtain speedup for difficult to parallelize codes.

2.7.2 Auto-tuners

The N-way approach is orthogonal to the one taken with auto-tuners. Auto-tuners usually rely on the presence of many tunable parameters for a given algorithm and will try to pick the best set of values for these parameters across a wide range of input data for a specific machine. Once the parameters are set, they do not change and that "version" is considered optimal for that machine. N-way exploits diversity when it is not clear a-priori which version will be better (ie: no fixed set of parameters is optimal). Auto-tuning could be used in conjunction with the N-way runtime to come up with a reduced set of good parameters which the runtime could take as inputs to configure the different ways. Note also that recent work on PetaBricks [3] similarly tries to trade-off execution time and quality of results but takes a more tuning approach as opposed to a competitive execution approach.

2.7.3 Isolation mechanism

The isolation guarantees provided by N-way are similar to those provided by STMs [32] but the mechanism to provide them is much more lightweight. No logging is required and N-way does not care about the interleaving of interactions. Versioned boxes [12] also use versioning to keep track of the different values of a variable but their approach seeks to solve a different problem than N-way. As such, their implementation is much more complex and slower than N-way's. Furthermore [12] provides isolation

in the context of STMs and not in the N-way context. Both approaches use some form of type wrapping to function.

2.8 Conclusion and future work

This chapter presented the N-way programming model as an alternative way to exploit parallel resources: instead of relying on a break-up of the computation, it exploits *algorithmic diversity* to obtain speedup or QoR improvements. In this model, a set of *ways* are run in parallel to solve a problem and the best one, for example the fastest, is picked just-in-time. This approach is justified by the demonstrated presence of *diversity* in algorithms (heuristics, parametrized algorithms and randomized algorithms in particular).

This thesis makes the following important contributions to the N-way model: **i)** a statistical learning model to estimate the *expected benefit* of sets of ways and therefore allow an *efficient* and *practical* use of N-way; **ii)** a culling framework based on the measure of progress in ways to further reclaim unproductive resources and **iii)** an API for N-way that simplifies state encapsulation and **iv)** a compiler flow for automated conversion of a C++ program to N-way incorporating state isolation.

2.8.1 Future work

Several extensions of the N-way work are possible. Firstly, the N-way model could be used to launch more ways than the available hardware parallelism. This would enable the runtime to have access to a wider array of possibilities for each way: instead of leaving it alive or killing it, the runtime could throttle its priority based on its performance.

Secondly, the N-way could be used in conjunction with traditional parallelism in which case N-way would act as a multiplier effect on traditional parallelism. Note also that different parallelization approaches could also provide another source of diversity.

Thirdly, the current N-way model throws away the $n-1$ ways that are not committed. However, if these ways have also executed for some time, they may have acquired useful information about the problem and/or the input. This information could possibly be incorporated into the main result thereby improving its quality. More generally, ways could collaborate during their execution, sharing tidbits of information about the problem being solved. This would violate the sandbox around the ways but if done in a controlled manner, this could prove beneficial.

Finally, since the N-way model relies on diversity to operate, it would be interesting to study how diversity could be directly measured. This would enable, for example, an alternate culling algorithm where the objective would be to cull ways that are similar thereby keeping the breadth of diversity intact while reducing the number of resources required to express it. Characterizing diversity between algorithms could also be used to further guide the runtime algorithm in picking which heuristics should be launched together and which should not: heuristics that are too similar should not be launched together whereas very different heuristics should as this helps maximize the diversity expressed.

2.8.2 Thesis discussion

The results for the N-way model demonstrate that an algorithmic property, *algorithmic diversity* can be usefully exploited to occupy idling cores and provide benefits both in terms of speedups and quality of results. Note that N-way makes no assumption on the parallel nature of the computation: it applies equally to hard to parallelize, sequential or embarrassingly parallel computations. The only requirement on the computation is that sufficient diversity exist.

The N-way model is therefore a non-traditional use of parallel codes to speculatively execute competing ways and pick the best one just in time. The N-way model also introduces culling to improve parallel efficiency. This culling works well when

a clear notion of algorithmic progress can be established. This notion of progress is again an algorithmic property that, if correctly expressed, can lead to better resource utilization. In practice, for greedy constructive algorithms, it is easy to define a notion of progress. Other types of algorithms which build sub-solutions that are later combined into a more complete solution are also good candidates.

Finally, note that the N-way model relies heavily on the notion of *diverse ways*. Measuring the amount of diversity is an open question. This thesis approaches it in part through the use of memory profiling as detailed in Chapter 4.

CHAPTER III

LEVERAGING DATA-STRUCTURE SEMANTICS FOR OPTIMISTIC PARALLELISM

As motivated in Section 1.2.1, irregular algorithms are difficult to parallelize. Current techniques, which include optimistic methods such as STMs, incur significant overhead due to the fact that they make mostly uninformed decisions about the data-footprints of the operations. This chapter proposes to leverage the semantic knowledge the programmer has about data-structures to help a smart runtime make better parallelization decisions. This work therefore is mostly concerned with improving the parallelization of irregular algorithms and not so much with the parallelization of hitherto sequential codes. It can however, in certain cases which are detailed in this chapter, help with traditional parallelization (data parallelism).

3.1 Data disjointedness

The key to parallelizing any computation is understanding the *disjointedness* between their data footprints: two computations with disjoint data footprints can be run in parallel. In this chapter, a computation is defined as being composed of **i)** an operation and **ii)** a data-extent or footprint. Mendez-Lojo et al. understood the importance of this view in [55] where they stress the importance of a *data-centric* view of a computation. They contend that instead of thinking about dependencies between operations, one must take a view that encompasses the actions of the operations on the data. As a reminder, $D_f(O)$ denotes the data footprint of an operation O and it conceptually contains information about all memory accesses of O.

In dense-matrix operations, $D_f(O)$ is easy to compute as dense-matrix operations

rely on indexing. A compiler can reason about the indices' ranges and statically determine whether or not parallelism is possible. This powerful reasoning ability has led to tremendous productivity improvements in the HPC domain. In irregular applications, $D_f(O)$ depends, more often than not, on the *runtime* value of variables which are not as easily bound as indices. Furthermore, the use of pointers complicates analysis as the variable from which to derive a value may be indirectly accessible through the runtime value of another variable.

This thesis hypothesizes that $D_f(O)$ can be viewed as a *semantic* representation of the data footprint of O and show that, given suitable programmer defined *predicates*, a runtime can dynamically reason about the disjointedness of data footprints of operations in irregular applications.

3.1.1 Disjointedness property

Disjointedness is really a property linked to both the *operation* and the *data* and the way they interact at runtime. For example, consider nodes and edges in a graph. No information can be inferred about the disjointedness of two operations O_1 and O_2 on distinct nodes n_1 and n_2 of the graph without knowledge of the semantics of the operations themselves. The fact that an 'edge' exists in the graph between n_1 and n_2 does not mean that O_1's and O_2's data footprints overlap. They will be disjoint if, for example, the operations do not refer to the neighbors of the nodes they are dealing with. The presence of a "link" in the data is therefore not an indication of a "link" between the data footprints. The association of the state of the data with the semantics of the operation is what determines the disjointedness property between two operations.

3.2 Opportunity in semantic information

Despite their lack of apparent static dependence structure, irregular algorithms often exhibit *semantic structure*. In other words, a programmer can frequently *estimate*

$D_f(O)$ based on very limited knowledge (the input to the operation for example). Consider the simple motivating example given in Figure 10. From the code, it is

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

Figure 10: Motivating example: a greedy graph coloring shown without any of the proposed abstractions

clear that the data footprint of an iteration of the loop on Line 4 is composed of the node `curNode` on Line 6, its neighbors in the graph and `lastColor` This information is easily obtainable given only `curNode` which is known at the beginning of the iteration. If the operation O is defined as one iteration in the main loop, a programmer who has knowledge of O's semantics can determine, at the start of the loop, O's footprint. While easy for the programmer to determine at runtime, this information is not statically known at compile time because the exact "neighbors" of a node are data dependent and depend on the runtime values of an adjacency matrix for example.

The combination of the *semantics* of the operation ("coloring") and the values of

```
1 struct statusColor_t {
        /* arbitrary status information */
  };

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

  DISJOINT(NODE_T v1, COLOR_F,
   NODE_T v2, COLOR_F) {
21   if(g.hasEdge(v1, v2)) {
       return 1.0;
     }
     return 0.0;
  }
26
  /* No DETERMINENEXT(COLOR_F) required */
```

Figure 11: Motivating example: a greedy graph coloring shown with the proposed abstractions (details in Section 3.4

an underlying data-structure (the adjacency matrix) form the basis for the decisions on data disjointedness. A different operation with different semantics but on the same data-structure would produce different disjointedness properties. For example, one could imagine a modified graph coloring algorithm that assigns a color not used by any neighbors and their neighbors. Although the data-structure would be the same, the semantics of the operation would change the disjointedness predicate. In both cases, the programmer's knowledge of the semantics of the operation is key to determining its data footprint.

3.2.1 Proposed approach: a semantic data footprint

This thesis proposes simple abstractions that allow the programmer to describe the *semantics of data access* of algorithms thereby enabling a runtime to determine whether two operations are disjoint and can therefore be run in parallel.

Traditionally, $D_f(O)$ contains memory locations touched by the operation O. However, this thesis argues that a *semantic* representation of $D_f(O)$ is more valuable. Consider for example, an operation that explores a planar space by picking a point at random and exploring a circle around it (a local search algorithm). A traditional data footprint of this algorithm would consist of all the memory locations of all the points touched in the search area. However, semantically, the programmer can represent this area with a tuple containing the center point of the circle as well as its radius. This representation is natural to the programmer as it flows from the semantics of the algorithm rather than the exact representation in terms of data structures. Representing the data footprint semantically has two main advantages. Firstly, it *incorporates the semantics of the data structure*. Instead of representing the footprint as an unstructured list of memory locations, it is represented in a semantically meaningful manner that the programmer can describe and reason about. Secondly, the *representation is much more compact*. In the example, representing the

footprint as a tuple is more efficient than representing it as a list of memory locations.

This new type of data footprint is called an *abstract data-space*.

3.2.1.1 Reasoning on the semantic data footprint

While convenient for the programmer, this high level representation of $D_f(O)$ makes it impossible for an application-agnostic runtime to reason about. In the traditional view of a data footprint, a runtime can always continuously construct the footprint (by tracking memory locations) and determine whether two footprints are disjoint (by computing the overlap between the memory locations). With abstract data-spaces, this is not possible and the programmer must therefore provide alternatives to both **i)** constructing the data-space and **ii)** determining disjointedness.

To address the first point and track the data-space's evolution, the programmer defines a `determineNext` predicate which computes an updated data-space based on the progress of the operation. This predicate is supported by methods provided by our API for the programmer to *track* this progress and make it available to the runtime in a non intrusive manner. `determineNext` also enables another interesting aspect of the system which is that the data-spaces need not be accurate but rather *estimates* that can be refined over time as more and more information about the computation becomes available. Consider for example an algorithm which searches a tree for a particular element and modifies it. Initially, the programmer only knows that the algorithm is potentially going to touch the entire tree but as it progresses, the area possibly touched by the algorithm diminishes. The more precise the estimation of the data-space, the more accurately the runtime can determine disjointedness.

To determine disjointedness, the programmer defines a `isDisjoint` predicate which the runtime calls to determine whether or not two data-spaces are disjoint.

Note that both predicates are specific to the operation O and can only be provided by the programmer.

3.2.1.2 Use-cases for the semantic data footprint

The semantic information provided by $D_f(O)$ can be used in two distinct contexts: **i)** dynamically extracting parallelism and **ii)** improving the performance of optimistic parallelism through a concurrency scaling mechanism. In the first case, the programmer writes a sequential application and parallelism is dynamically extracted where possible. In the second case, the original application is already parallelized using optimistic techniques (such as software transactional memories (STMs) [33]) and performance is improved.

Parallelization In many parallel models, a `foreach` construct exists where each loop iteration can execute in parallel. This construct is very similar to the `DOALL` construct in FORTRAN except that the iterations can be statically checked for overlaps [8].

The proposed approach uses the dynamic semantic knowledge of $D_f(O)$ to determine whether an iteration O is disjoint from all other concurrent iterations and can therefore be launched or if it should be launched at a later point. When launched in parallel, no locks are used as the framework guarantees that the operations will not conflict. Note that in this case, the $D_f(O)$ for every operation O needs to be a sound *over-approximation* of the actual data-space of O.

This use-case therefore allows the parallelization of sequential code although the type of parallelism exposed is the traditional data-parallelism.

Throttling concurrency STMs are a natural fit for irregular applications as they eliminate the need for a programmer to ascertain the disjointedness of two operations. Instead, the programmer simply has to specify transactions that must execute atomically and let the STM runtime ensure that all transactions are indeed atomic. While greatly simplifying the programming of irregular algorithms, STMs suffer from

severe overheads particularly in the case of frequent rollbacks. $D_f(O)$ can be used to *throttle* the start of transactions to ensure that they have a low probability of conflict. This leads to fewer rollbacks, which results show can be reduced by up to 80 %, and lead to significant performance improvements in the STAMP benchmarks. The `determineNext` predicate is important in allowing running transactions to update their $D_f(O)$ and therefore allow a more accurate detection of overlap particularly for long running transactions which is particularly interesting as long running transactions are usually avoided in STM systems due to their high cost in case of failure.

3.3 Data-structure semantics

Given an operation O_1 operating on an input I_1 in an irregular application, this work seeks to

- reason on $D_f(O_1)$ to determine its disjointedness with other operations
- reason about its evolution over time as O_1 progresses.

The `isDisjoint` predicate addresses the former concern while the `determineNext` predicate addresses the latter. An example of the proposed constructs is shown in Figure 11. Changes from the original code (Figure 10) are minimal and mostly involve defining tags and operations (Lines 5 and 6) as well as providing a `isDisjoint` predicate (Line 20). This section introduces the predicates and constructs and detail their use in the example in Section 3.4.

For both predicates, the programmer only reasons about semantic objects (as opposed to memory locations) which makes it practical for him to write them.

3.3.1 Disjointedness predicate

This predicate is used to express the programmer's *semantic knowledge* of how objects in the data-space relate to the operations operating on them.

In the example of the local search in a plane, the disjointedness predicate would express the fact that tuple (C_1, R_1) and tuple (C_2, R_2) are disjoint if and only if $\text{dist}(C_1, C_2) > R_1 + R_2$. In the graph coloring example in Figure 10, the disjointedness predicate would simply determine if there is an edge between the two nodes given as input to the iterations. In both cases, the condition is only derivable from the semantic knowledge of how the operation acts on the underlying data.

The disjointedness predicate is defined for pairs of elements as follows.

Definition Let O_1 and O_2 be two distinct operations, for any element a_i in $D_f(O_1)$ and any element b_j in $D_f(O_1)$, the pair-wise disjointedness of (O_1, a_i) and (O_2, b_j) is defined as a function that returns the likelihood of whether the two elements will result in overlapping memory accesses.

$D_f(O_1)$ and $D_f(O_2)$ overlap if and only if some a_i overlaps with some b_i. $D_f(O_1)$ and $D_f(O_2)$ are said to be fully disjoint if every pair is disjoint. This predicate is referred to as $\text{Pred}_{\text{D}}(O_1, a_i, O_2, b_j)$.

3.3.2 Determine-next predicate

The determine-next predicate serves to refine the data-space of an operation O_1. While initially, $D_f(O_1)$ is based only on the knowledge of its input I_1, as O_1 progresses, more and more information about O_1 is available and its data-space can be made more accurate. This is an optional predicate that can be used to improve the accuracy but is not required to determine disjointedness.

For example, in Figure 10 there is no need for the `determineNext` predicate as the footprint can be accurately described using just the input to the operation.

The determine-next predicate takes as input a structure tracking the *state* of O_1 as defined by the programmer and returns a new data-space for O_1. In other words, at the start of a transaction, $D_f(O_1)$ is determined solely based on its input I_1 but, as O_1 progresses, a new semantic $D_f(O_1)$ can be computed based on the state of O_1.

The state of the operation O_1 is entirely defined by the programmer and can be used to track key values that determine future memory accesses. This predicate is referred to as $\text{Pred}_{\text{DN}}(O_1)$.

Note that since Pred_{DN} can generate a whole new $D_f(O_1)$, the specific Pred_{D} that will be used to determine disjointedness between O_1 and other concurrent operations can change during the execution of O_1 based on the value of its state. The solution proposed is therefore very flexible and can be applied to arbitrarily complex operations that can have multiple "stages" (captured in their state).

3.3.3 Specification

This section defines the concepts needed to allow the programmer to utilize the two predicates introduced above: `OperationRole` and `ValueTag`. With these concepts, the programmer can identify the *semantic types* of functions as well as their associated *status* (`OperationRole`) and data (`ValueTag`). This is crucial as the disjointedness predicate links both of these concepts in ways that are specific to each `OperationRole` and `ValueTag`.

3.3.3.1 `OperationRole`: Encapsulating operation semantics

An `OperationRole` is an identifier given to a set of operation semantics such as "traverse depth-first", or "traverse breadth-first" or even simply "traverse". An `OperationRole` can be viewed as a tag that identifies the function to the runtime system. Different functions may have the same `OperationRole` if they have the same semantic role. The programmer can associate an `OperationStatus` data structure which will serve as input to $Pred_{DN}$ predicates. The `OperationStatus` will be made available by the runtime to the function and can be updated throughout its execution to report on its status.

3.3.3.2 `ValueTag`: A semantic tagging system

Similarly to the `OperationRole`, values also play a "role" in the predicates: an operation may act differently on the initial node and its children node. The C/C++ type system is not concerned with distinctions based on the role of a value in the program and is therefore inadequate in capturing the semantic meaning of a value. Furthermore, dynamically, the role of the same value may change and a static type system cannot capture this. The `ValueTag` dynamic tagging abstraction solves this.

Consider again the example in Figure 10. The original node used as input and the neighbor nodes used inside the computation play different parts in the determination of the predicates. Indeed, only the original node is used to determine disjointedness with other operations whereas Pred_D is not called for any of the children nodes. As far as C/C++ is concerned, they are all pointers but they have a semantic meaning associated with the computation. This semantic meaning can change as time progresses and a tag instead of a type addresses this issues. Therefore, for a single C/C++ type, multiple "roles" may exist and this is captured by the `ValueTag`. Conversely, a single `ValueTag` may also encompass different C/C++ types.

3.3.3.3 Formal definition of the predicates

With these two concepts in hand, the definition of Pred_D and Pred_DN can be formalized.

Definition $\mathcal{S}_\mathcal{O}$ is defined as the space of `OperationRole` and $\mathcal{S}_\mathcal{V}$ as the space of `ValueTag`. The set $\mathcal{S}_\mathcal{S}$ is the space of `OperationStatus` and is bijective to $\mathcal{S}_\mathcal{O}$. In practice, they are finite sets of elements.

Definition of Pred_D Pred_D follows the prototype given in Equation 1.

$$(\mathcal{S}_\mathcal{O}, \mathcal{S}_\mathcal{V})^2 \longrightarrow [0; 1] \tag{1}$$

It respects the following properties:

- $\text{Pred}_\text{D} = 0$ if and only if the input `ValueTags` used in their respective `OperationRoles` will not result in accesses that will conflict (no write and read or write and write to the same location).
- Conversely, $\text{Pred}_\text{D} = 1$ if and only if there will be a conflict.
- In all other cases, the result is a value between 0 and 1 which represents the likelihood of conflict.
- Pred_D is symmetric and $\text{Pred}_\text{D}\,(O_1, a_1, O_1, a_1) = 1$

Definition of Pred_DN Pred_DN follows the prototype given in Equation 2. It returns the new footprint for the operation (containing n elements).

$$(\mathcal{S}_\mathcal{S}) \longrightarrow \mathcal{V}^n \tag{2}$$

The programmer can thus specify a series of such predicates. At runtime, when the `ValueTypes` and `OperationRoles` are known, the selection of the applicable predicates will use the following rules in order:

- If an exact match on the `ValueTypes` and `OperationRoles` is available, that predicate applies
- Otherwise, a lexicographical traversal of the hierarchy of the `OperationRole` `ValueType` pair is used. By lexicographical, we mean: "dc" → "db" → "da" → "cc" → "cb" ...
- If no match is found, the predicate is deemed not to exist and will therefore default to not returning anything for Pred_DN and returning 0 for Pred_D.

The effect is to pick the most specialized predicate in the hierarchy. The hierarchy gives the flexibility to the programmer to define predicates that are true for a wide array of very specialized `OperationRoles` or `ValueTypes`.

3.4 Runtime implementation

Section 3.3 formally defined Pred_{D} and Pred_{DN}. This section describes how these concepts translate into C++ code and how the runtime makes use of them. The described framework is implemented in C++ due to the wide availability of accepted benchmarks in C and the power of template meta-programming in C++ (crucial to an efficient implementation of the runtime). Most of the visible API is defined as macros that hide the actual complexity of template meta-programming.

This section also describes the role of the runtime in efficiently making use of the predicate information. Finally, it presents how the runtime improves both parallelization and successfully throttles transactions to lower their abort rate and improve their execution time.

3.4.1 Programmer specifications

The code given in Figure 11 is used as an example; it is very close to the actual implementation which shows that the burden on the programmer is light.

3.4.1.1 Specifying the roles and value tags

The programmer is responsible for enumerating the semantic types he is interested in tracking. This task is relatively simple and follows from the design of the algorithm itself. Each operation can be statically annotated by its `OperationRole` and each variable can be dynamically tagged (as in, the tag may depend on the control flow) by its `ValueTag`. Note that `OperationRoles` are considered to be scoped with the most deeply nested one being the active one. In Figure 11 the `OperationRole` is defined on Line 6. In this simple example there is only one `OperationRole` but there is no fundamental reason why this should always be the case. The programmer also associates an arbitrary data structure to each `OperationRole` which serves as the `OperationStatus` associated with that `OperationRole`. In this case, there is no need for a status variable and therefore the `statusColor_t` structure is empty.

The `ValueTag` is identified on Line 5. Note that a `ValueTag` is associated with a single C/C++ type. The reverse is not required though as the same variable may be tagged with different `ValueTags` at different times in the program. The restriction to a single C/C++ type for a `ValueTag` is due to an implementation detail but does not harm functionality in practice.

The `curNode` variable is tagged with the `NODE_T ValueTag` on Line 12. Although tagging seems static, the framework allows the dynamic tagging of variables: the exact tag may depend on the control flow of the program. This indicates to the runtime that `curNode` has a value type of `NODE_T` which will determine which predicates that be applied to it.

3.4.1.2 Specifying the predicates

Now that the user has defined $\mathcal{S}_{\mathcal{O}}$ and $\mathcal{S}_{\mathcal{V}}$, the predicate functions are functions that are distinguished based on their input `OperatorRoles` and `ValueTags`. The programmer only needs to define those that make sense to him. A default function (indicating that no overlap exists) is provided for all combinations of $\mathcal{S}_{\mathcal{O}}$ and $\mathcal{S}_{\mathcal{V}}$ that is not defined. In the example, this is defined on Line 20: the programmer defines Pred_{D} which determines whether or not an edge exists between the two `NODE_T` (called `v1` and `v2` in the code).

3.4.1.3 Specifying the operation

The programmer needs to identify the operations, which, in STM semantics, can be viewed as the atomic sections. This is shown on Line 13. The `OperationRole` is defined as well as its initial input. This specification will bind the predicates that are applicable to the `OperationRole` of the operation (here `COLOR_F`) and the `ValueTag` of the input variable (here `NODE_T`). The extent of the operation is also indicated with the curly braces and the code within the operation does not need to change as it will either be launched with no concurrency overhead or sequentially in case a conflict is

detected.

3.4.1.4 Summary

To review, the programmer is required to define **i)** the space of the `OperationRoles` and the `ValueTags`, **ii)** the predicates, **iii)** the `ValueTag` for variables that are used as input to operations and **iv)** the operations themselves.

Provided the programmer understands the semantics of his program, these requirements are easily met. As showed in the experiments, for many benchmarks, the specification of these requirements is simple and straightforward. The flexibility of the framework allows it to be applicable to a wide range of programs.

3.4.2 Low-overhead runtime

The runtime is implemented in C++ and makes heavy use of templating mechanisms to allow compile-time selection of the appropriate predicate function for all combinations of values from $\mathcal{S}_{\mathcal{O}}$ and $\mathcal{S}_{\mathcal{V}}$. This significantly improves the runtime binding operations for these predicate functions. Note that while the runtime is in C++, any code written in C or C++ can make use of it. In the experiments, the STAMP benchmarks which were written in C were used.

The main role of the runtime is to apply the appropriate predicates when an operation is being launched and to determine whether or not it can concurrently execute with the other currently executing operations. The runtime will follow Algorithm 1 to best determine the likelihood of a conflict between an operation that is launching and those that are already running. The `overlapLikelihood` is a number between 0 and 1 where 1 means that an overlap is certain.

3.4.2.1 Degree of approximation

The runtime will also estimate how much time it should spend checking for overlaps. This is particularly important if there are many concurrent operations as a new

```
Input: initialElement the initial element passed to the operation
Input: currentOperation the operation that is starting
Output: overlapLikelihood whether or not the operation will conflict (to the
        best of the runtime's knowledge
overlapLikelihood ← 0 ;
foreach ConcurrentOperation concOp do
    concurrentDataspace ← DetermineNext (concOp) ;
    foreach elt in concurrentDataspace do
        res ← IsDisjoint (currentOperation, initialElement, concOp, elt) ;
        overlapLikelihood ← overlapLikelihood + res ;
    end
end
return overlapLikelihood;
```

Algorithm 1: Simplified runtime algorithm to determine the likelihood of conflict between the operation to launch `currentOperation` and existing operations.

incoming operation has to be tested against all existing operations. The intuition behind measuring runtime overhead is that the time required to check for disjointedness should not be significantly more than the time wasted by not running in parallel or by running in parallel and having to abort (in a STM system). Therefore, the runtime maintains statistics about the execution of the operations and uses them to determine how much time it can use up to check for disjointedness. It will then bail out of Algorithm 1 whenever it has spent at least that much time performing checks.

Monitoring for the parallelization approach In the simplest case, the runtime has the choice between running an operation in parallel with no concurrency checks or running sequentially (in case of a conflict). Here, the time that the runtime keeps track of is the average time required for an operation. Indeed, that is the "cost" of not running it in parallel (as it will have to be serialized). The runtime will then perform checks for a small user-defined fraction F of that time. If within that time it successfully completes Algorithm 1 and determines that the `overlapLikelihood` is 0, it will launch the operation in parallel; otherwise, it will serialize it.

Monitoring for the throttling approach In this case, the "cost" of making a bad decision is a rollback. The cost of the rollback includes the cost to partially execute the transaction (an operation here) and the overhead of the actual rollback mechanism. However, a rollback does not always occur so on average, the cost incurred is $\frac{T_A}{C_C+C_A}$[1] where T_A is the total amount of time that the transaction spent aborting, C_C is the commit count and C_A is the abort count. This quantity captures both the likelihood of a rollback and the average cost of that rollback. If no rollback has ever occurred, the runtime will not perform any checks and just let the transaction run as it is highly likely that there will not be any conflict (given past history). Again, the runtime will perform checks for a fraction F of this quantity.

3.4.3 Runtime usage

The goal for this work is to **i)** improve parallelization and **ii)** improve the performance of transactional systems through the throttling of their transactions. This section explains how the runtime is used to implement these goals.

3.4.3.1 Improving parallelization

In this context, the runtime has to be positive that disjointedness is maintained since the goal is to launch the operations in parallel with no runtime checks (such as STMs). For two operations O_1 and O_2, given that $D_f(O_1)$ and $D_f(O_2)$ are over-approximations of the actual data-spaces, if the programmer provides the appropriate $Pred_D$ predicates, the runtime will only launch O_1 and O_2 in parallel if `overlapLikelihood` is exactly 0. This condition is constraining and may not be possible to check in a small amount of time but, as Section 3.5 demonstrates, for the graph coloring example, this condition can be easily checked.

If the runtime check cannot conclusively determine disjointedness, the safe option

[1]Here, it is assumed that other transactions could have run had the transaction that rolls back not run.

is chosen and the operation will be serialized. Note that since the time of the runtime check is limited to a small user-defined fraction F of the average sequential time of the operation, in the worse case where no parallelism can be safely found, a slowdown is incurred over the sequential execution that is completely determined by F.

3.4.3.2 Throttling transactions

In this case the level of certainty about the disjointedness of two operations does not need to be as high as the STM system provides a "safety net"; only a reduction in the number of aborts is sought. The runtime computes the sum of `overlapLikelihood` between $D_f(O_1)$ and the data-spaces of all other concurrently running transactions. If this sum is higher than a certain threshold, O_1 will be paused until the transaction with which it has the highest `overlapLikelihood` completes. The programmer can therefore modify both the fraction of time to check (F) and the probability threshold. In the previous scenario, that threshold was 0 (ie: everything had to be fully disjoint). A higher threshold increases the potential amount of parallelism but can also lead to more conflicts.

3.5 Experimental evaluation

This section demonstrates the benefits of this work through a simple greedy graph coloring algorithm as well as through several STAMP benchmarks to illustrate its wide applicability.

All experiments were performed on a dual quad-core Intel Xeon E5540 (2.53GHz) with up to 8 concurrent threads. The number of threads was not increased past this limit to remove the issues related to kernel level thread scheduling

3.5.1 Greedy graph coloring

The greedy graph coloring algorithm was introduced in Section 3.2. Section 3.3 explained the applicability of the two predicates Pred_{D} and Pred_{DN}. It also discussed

how it is extremely simple for the programmer to specify a Pred_D predicate which determines if $D_f(O_1)$ and $D_f(O_2)$ are disjoint. All the predicate needs to do is determine if there is an edge between two inputs I_1 and I_2.

Two versions of the algorithm (Figures 10 and 11) were implemented in C++.

The first was a traditional parallel version of the algorithm. The processing of each node was assigned to a different task. A STM (tinySTM [26]) was used to wrap the critical section using atomics. The `parallel_for` construct in TBB [67] was used to launch all the tasks in parallel. TBB also managed the mappings of tasks to the underlying threads. This is the baseline case.

The second was a parallel version which applied the proposed approach. This version created a number of TBB tasks and scheduled them based on the execution of the Pred_D predicate. Instead of blindly selecting the next task from the run queue, the scheduler postponed the execution of any task which was not disjoint with the other tasks that were currently running (through the evaluation of the Pred_D predicate). Indeed, the implementation could have waited until the conflicting task could be scheduled again and spun in a tight loop until that time. However, by postponing the execution and choosing another disjoint task makes much better use of the available parallelism. In this version, the use of any locks or STMs around the data structures was avoided, it is guaranteed that the execution of any two concurrent tasks would not have an overlapping data footprint.

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

Note that the proposed approach is particularly beneficial for long running transactions. To study the behavior of the system in the presence of long running transactions, transactions of different durations were simulated and results show that as the duration of the transaction increased the speedup achieved increases. This is expected, since as the duration of a transaction increases, its associated cost of rollback

(in the case of a conflict) correspondingly increases. Hence, assurance of no conflicts before launching a transaction dramatically increases performance. Figure 12 shows

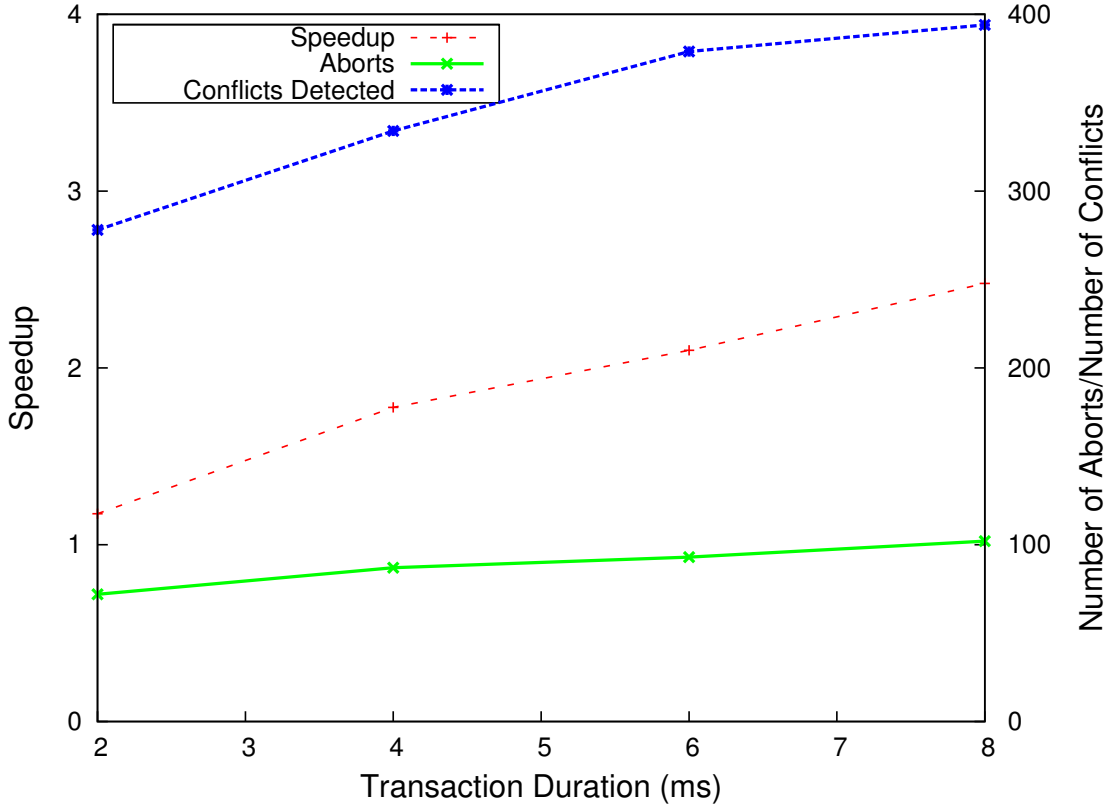


Figure 12: The Greedy Graph Coloring Benchmark

the performance of the system with transactions of varying durations. The duration of transactions was varied by introducing appropriate sleep calls in the transaction. The speedup line indicates the ratio of time it takes to execute a traditional parallel implementation to an implementation which uses the proposed approach (in other words, this is not the speedup relative to a sequential implementation but rather expresses the benefit of the proposed method over a traditional parallel implementation). The "Number of aborts" is the number of aborts that the STM system had to perform (in the case of the traditional parallel version). The "Number of conflicts" is the number of times the proposed scheduler detected a conflict in the data-spaces through the execution of the Pred_{D} predicate and decided to postpone the execution of the task (in the case of the enhanced parallel version).

With a transaction duration of 8ms, the system was able to provide a speedup of over 200% over a traditional parallel implementation. The "Number of conflicts" increases with the duration of the transaction since there is more potential for conflict. The "Number of aborts" increases as the number of threads increases as expected. This is because the amount of contention increases and the STM system needs to rollback more frequently. Note, that the total cost of aborting is proportional to the number of aborts and the duration of a transaction.

3.5.2 STAMP benchmarks

Several STAMP benchmarks were also modified to take advantage of the proposed approach by adding simple predicates. In particular, the KMeans (K-means clustering), the Yada (Delaunay mesh refinement; Ruppert's algorithm) and the Labyrinth (maze routing) benchmarks were modified. The benchmarks chosen had long transaction lengths and large read/write sets. Such benchmarks are most amenable to the proposed approach since the runtime can provide significant performance improvements. All experiments were performed on a 8-core machine with 8 concurrent threads. Other STAMP benchmarks would not benefit as much from the proposed approach as their transactions are much shorter.

3.5.2.1 K-Means

Transactions in the K-Means benchmark write to a shared array. Conflicts arose due to the fact that several transactions could potentially be writing into the same parts of the array at the same time. A STM system rolls back transactions in the case of such a conflict. An extremely simple Pred_D predicate, which determines if two transactions will be accessing disjoint parts of the array or not, was added. The predicate is simply an equality comparison of the indices that each of the transactions will be accessing. It is important to emphasize how simple it is to write this Pred_D predicate (here just one line of code). In this case, 0 is returned if there is no overlap

and 1 if there is.

3.5.2.2 Yada

Transactions in the Yada benchmark (based on the Delaunay thread refinement algorithm) try to "refine" elements by working in a cavity (a neighborhood) around themselves. Each refinement reads and writes to that cavity and if two elements being processed concurrently are too close to each other a rollback will occur. A simple Pred_D predicate which compares the distance from the center point of the elements to the sum of their radius was added. If the distance between the two elements is larger than a small multiple of the sum of their radius, the transactions are considered to be disjoint and allowed to proceed. In the other case, one transaction is halted to give the other time to finish and avoid an expensive rollback. 0 is returned in case of no conflict and 1 in case of a potential conflict.

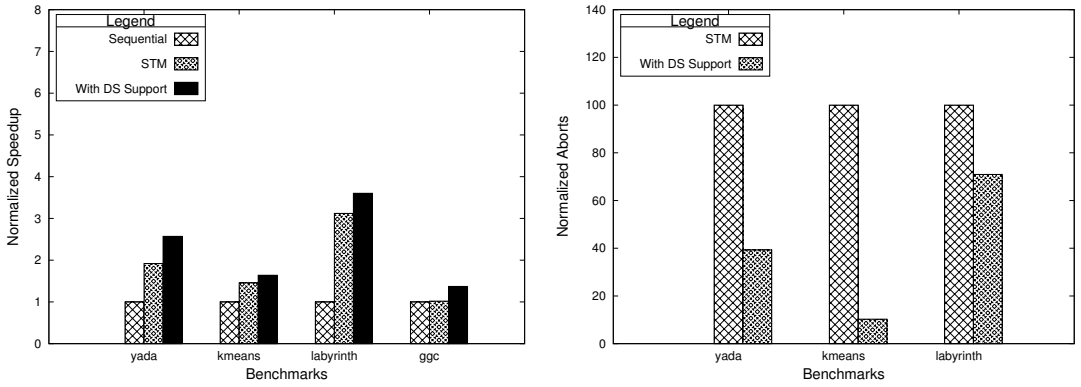
3.5.2.3 Labyrinth

In the labyrinth benchmark, the transactions use Lee's routing algorithm to find the shortest path between a pair of nodes on a 3D grid. Once the shortest path is found, the transactions write back the path to the grid thereby causing a conflict if a concurrent transaction wants to write to the same cell in the grid. To reduce the number of aborts and improve the execution time, a very conservative Pred_D predicate that compared the spanning cube of the source and destination points of the transactions was implemented. If the cubes overlapped, the transactions are considered to have a potential to overlap, otherwise they are not and are allowed to proceed. Note that here a fractional `overlapLikelihood` based on the overlap volume compared to the total volume of the cubes is returned. This benchmark also uses the Pred_DN predicate to periodically update the size of the spanning cube during the execution of the transaction. Once the shortest path is found, and the transaction is writing back the path, the size of the spanning cube is reduced periodically, to

encompass only the region which will be written to in the future. Note that though this approach is extremely conservative and could potentially lead to long waits if the cubes are too big, for more numerous and smaller paths, the approach worked very well especially due to the ability of the runtime to refine approximations through the Pred_{DN} predicate.

3.5.2.4 Results

Figure 13 reports the speedups achieved in the experiments as well as the reduction in the number of aborts. Speedups are relative to a sequential execution of the benchmarks. Note that in all cases the number of aborts drops significantly and the



(a) Speedups obtained in different benchmarks (b) Normalized number of aborts in different benchmarks

Figure 13: STAMP Benchmarks

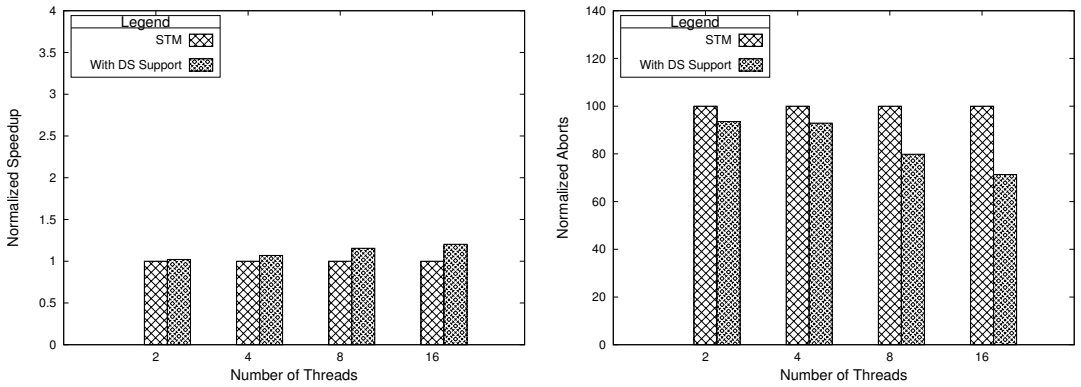
performance improves significantly given the low-effort requirements for the programmer. This technique therefore has a very low cost to benefit ratio providing very good results at a very low cost for the programmer. Note that the fact that aborts still exists is because of the approximate nature of the runtime. Indeed, it does not always fully check the predicates if it considers that it is not worthwhile to do so (based on the cost of previous rollbacks). The fact that the number of aborts goes down so dramatically makes the system particularly well suited for long running transactions where the cost of an abort is high (in particular for transactional systems that use

commit time locking).

3.5.3 Scaling

A more detailed analysis of the Labyrinth benchmark was performed and the benchmark was run with 2 to 16 threads. The results show both a drop in the number of aborts and an increase in performance, the magnitude of which increased as the number of threads was increased.

Therefore, as the number of threads increases the system provides excellent scalability as compared to traditional approaches and such an approach is essential in the presence of a larger number of concurrent threads. Figure 14 reports the results obtained my running the Labyrinth benchmark with a varying number of threads.



(a) Normalized Speedups obtained in the Labyrinth benchmark

(b) Normalized number of aborts in the Labyrinth benchmark

Figure 14: Labyrinth Benchmark

3.5.4 Impact of limited check time

An important characteristic of the runtime is that the amount of time spent checking for overlaps is capped. Whether this impacted performance was investigated by varying the user-defined fraction F defined in Section 3.4.2.1 which controls the amount of time the runtime will spend checking: for 16 threads for the Labyrinth benchmark,

all checks occurred with $F \geq 0.2$ and for K-Means, all checks occurred with $F \geq 0.8$. In both cases, F can be smaller than 1 which means that the system will be able to extract a performance benefit. The higher required value for K-Means can be explained by the fact that a K-Means transaction is shorter than a Labyrinth one and since the time allowed to check is F times a quantity related to the abort cost, F needs to be high enough to give the runtime sufficient time.

For F smaller than those values, the runtime did not cause a performance degradation and was not always able to improve performance. This was due to it not being to catch as many possible conflicts and therefore letting transactions proceed that would ultimately abort.

3.6 Related work

The Galois programming model [46, 55] also takes a data-centric semantic approach to detecting conflicts. In the Galois model, the programmer defines whether or not two operations can *commute*: if they can, the two operations can operate in parallel and if they cannot, the two operations must be serialized. Although the premise for both works, utilizing semantics information to improve parallel execution, is similar, the work described in this chapter differs from the Galois model in several ways.

First, the Galois model focuses on semantic information at a much lower granularity: at the level of the individual operation on a specific data structure rather than an entire algorithm. Indeed, they demonstrate how defining commutativity properties for functions such as "add", "delete" on a list can allow them to detect conflicts. Although one could imagine applying the Galois model to whole algorithms, this does not seem to be the primary intent of the model. As such, the Galois model lacks concepts similar to `determineNext` and `OperationStatus` which track the *evolution* of data footprints during the execution of the algorithm.

Secondly, the execution model for the Galois model is to use semantic information

to detect conflicts but not to prevent them. The described model seeks to *predict* the likelihood of a conflict and *prevent* it from happening altogether by preventing conflicting transactions from executing concurrently. The Galois model, on the other hand, detects conflicts *during* the execution of a transaction and therefore has no choice but to trigger a rollback. The rollback triggered is slightly different from regular STM rollbacks as the model allows for a semantic rollback instead of a purely memory-based rollback but this still incurs significant costs. To alleviate this problem, [55] does describe the "one-shot" optimization which seeks to over-approximate the footprint of an operation and grab locks on all objects in the footprint *before* the operation's execution. However, unlike the described model, the one-shot optimization is only applicable to algorithms where it is possible to accurately approximate the footprint at the start. The use of the `determineNext` predicate allows for better approximations of the footprint as the algorithm progresses. This also makes for a less conservative as no locks are held; a large over-approximation with the one-shot optimization can potentially prevent many other threads from advancing even if there is no conflict.

Apart from the Galois approach, two major directions have been looked at to extract parallelism from amorphous code: a static approach and a dynamic approach.

3.6.1 Static extraction of parallelism

Dynamic Parallel Java (DPJ) [8] is a language that provides a type and effect system that is verifiable at compile time allowing the compiler to make strong assertions about the effects of a particular operation on the data. The strong guarantees that DPJ makes about a program allow it to make some very interesting static optimizations like eliminating some of the concurrent overhead of locks or transactions when it can be statically certain that no conflict is possible. The work described in this chapter is orthogonal to DPJ as it relaxes the guarantees that are made to the programmer but enables the detection of more potential parallelism (where, for example, parallelism is

dependent on the *value* of a variable as opposed to its extended type in DPJ). It would also be interesting to combine both frameworks to provide some static assertions and allow the knowledge gain at compile time to aid in the dynamic runtime predicate evaluation. Work by Reid et al. [66] is similar work which also extends the type system by using a notion of static ownership of data. Each data object is *owned* by a logical owner (a parent class, a global owner known as "world" for global objects or any other object in the scope) and an effect system is also introduced for functions. Static reasoning similar to DPJ can then be used to perform automatic parallelization.

3.6.2 Dynamic extraction of parallelism

Another approach that has been explored is the dynamic detection of parallelism such as Rinard's Jade [69]. This work is closely related as it also dynamically annotates data structures and dynamically evaluates conditions to determine the potential overlap of operations. However, whereas Jade focuses on identifying conflicts at a byte level, this work utilizes semantic information and the programmer's knowledge about an operation to determine whether or not two operations overlap. The footprint of an operation is also allowed to be refined over time as more and more information about the operation becomes available. The `determineNext` predicate is also unique in its tracking of the evolution of an operation. Work on serializer sets by Allen et al. [2] also seeks to dynamically extract parallelism by having the user specify *serializers* which correspond to operations that conflict with one another and therefore must be executed sequentially. This work is interesting because it also brings in some semantic knowledge about the operation to build the serializers however the approach taken is very different from the one described here.

A lot of work has gone into STMs, in particular to reduce the overheads of the STM runtime. Some recent work [80] attempts to reduce the overhead of STMs by replacing atomic sections by locked sections when appropriate. Many other techniques have also

been developed (see [48] for a recent list). However, this work does not specifically target STMs as such but rather STMs can benefit from the analysis led here.

3.7 Conclusion

This chapter motivated the fact that the disjointedness of two operations is really an intricate dynamic relationship between the operation and the runtime values of the data. Therefore, static approaches will not always be able to capture all the expressible parallelism in a program. Static approaches do provide the comfort of being certain that the execution is correct but are necessarily overly conservative about potential parallelism. The described approach allows the programmer to define some very simple semantic predicates that will get executed at runtime to determine the disjointedness of two operations. The `determineNext` predicate is particularly novel in allowing the programmer to express the *progress* of a computation and how this impacts its data-space (footprint).

Communicating this very simple knowledge that the programmer has about his algorithm shows promising results. Results show that the framework can be used to parallelize certain sequential algorithms in an efficient manner without incurring the overheads of a transactional system. Furthermore, for algorithms already utilizing a transactional system, results show that their performance can be improved by accurately predicting whether or not a transaction will abort. The reduction in the number of aborts translates to a reduction in execution time.

3.7.1 Thesis discussion

This chapter shows that the performance of optimistic parallelism to parallelize hard to parallelize irregular algorithms can can improved through the expression of data-structure semantics. The framework presented in this chapter therefore does not increase the utilization of cores but rather increases the *efficient* utilization of those

same cores: limiting the number of rollbacks contributes to less time spent on computation that will be ultimately be thrown away and re-executed. Indeed, optimism is a means to increase parallelism however, while aggressive optimism increases the amount of parallelism and core utilization it can also be be detrimental as it can lead to a large number of rollbacks. On the other hand, a guided form of concurrency such as the one proposed in this chapter can ensure that optimism is useful. This chapter shows how potential conflicts can be determined via the algorithmic properties on memory footprints. Such properties are then quantified via a probability measure that is used in scheduling the concurrent optimistic operations in parallel.

More generally, however, a better understanding of the data footprint of an operation can lead to better parallelization opportunities.

This chapter demonstrated that leveraging additional semantic knowledge of the application, known to the programmer but hidden or obfuscated in the program and to the compiler provides significant benefit. This furthers the claim made by this thesis that algorithmic properties can be effectively exploited to improve parallelization of hard to parallelize codes. In particular, this chapter shows that even for codes that are considered "parallel" (through the use of optimistic parallelism) further improvements are possible.

Chapter 4 further explores automatic ways to extract the semantic relationships in data-structures. Although not all semantic relationships can be deduced, Chapter 4 shows that the use of a profiler in the *semantic* memory space of a program can extract certain semantic relationships thereby enabling the *automatic* discovery of the neighborhood of an operation.

CHAPTER IV

DISCOVERING OPTIMISTIC DATA-STRUCTURE ORIENTED PARALLELISM

As described in Chapter 3, the semantics of data-structures can lead to performance improvements for optimistic parallelism. This improvement comes from the fact that the semantics associated with data-structures allows a runtime to make statistical predictions about future data accesses.

The previous chapter assumed that the programmer has a certain knowledge of his application and is therefore capable of writing accurate predicates. However, in some cases, the programmer may not completely understand the application and may therefore not be in a position to determine what the `determineNext` and `isDisjoint` predicates should be. While it is not possible, in general, to fully replace a programmer's knowledge about an application, this chapter presents a profiling based technique capable of extracting certain data-structure semantics information from a program thereby enabling the *automatic generation* of `isDisjoint` functions.

Note that the utility of the work presented in this chapter is not limited to being applied to the work in Chapter 3. In particular, in [63], Pingali talks about the importance of "determin[ing] the neighborhoods" of operations. This is very similar, in concept, to the `isDisjoint` function. Also, this work is more widely applicable in understanding access patterns at a higher level of abstraction than the low memory address level.

4.1 Address dataspace versus symbolic dataspace

The approach presented here is an automated solution to determine `isDisjoint` functions. It is based on a profiling approach: the framework seeks to determine the extent of an operation by profiling its loads and stores. Ultimately, it is interested in determining the extent of an operation before the operation starts to be able to determine whether or not it can run in parallel with other operations. Unfortunately, in a pointer-based algorithm, this type of profiling will produce a profile that is only exploitable for that run of the application (in other words, it has no predictive power for other runs). This is because pointers obfuscate memory patterns and offsets between memory locations (the only "relationships" that can be deduced from such a profiler) are unstable between runs and do not provide any meaningful information (except maybe on the inner workings of the memory allocation algorithm).

Therefore, this work proposes to profile in the *symbolic* dataspace of the program. Briefly, the symbolic dataspace is a dataspace where program variables are associated with programmer given *names*. For example, if variable `n` is located at address `0xdeadbeef`, a load to `n` would be profiled as `Load 0xdeadbeef` in a memory dataspace profiler but would be profiled as `Load "n"` in a symbolic dataspace profiler.

One way to effectively parallelize code is to understand its data access patterns: this allows a compiler or runtime to *predict* future accesses and therefore determine whether or not two sections of code will access the same memory location and therefore prevent parallelization. In dense array computations, data access patterns can frequently be understood statically at compile time by analyzing both the indices and the boundary conditions (loop bounds) used to access data. In the simplistic example loop shown in Figure 15, a compiler can predict the exact memory locations that will be accessed at the start of each loop iteration. These locations are solely determined by: **i)** the address of `dataArray`, **ii)** the address of `copyArray` and **iii)** the value of `i`; the values of which can be determined at the start of each loop iteration. In

this example, the memory locations will be `&dataArray + i` and `&copyArray + i`. Furthermore, with the runtime knowledge of `end`, all accesses during the loop can be predicted. Therefore, for dense array computations, predictions can be made directly

```
for(int i=0, e=end; i < e; ++i) {
  dataArray[i] = copyArray[i+10];
}
```

Figure 15: Example of a simple loop for a dense array where memory access patterns are fully predictable at the start of the loop

in the address dataspace which corresponds to the mapping of variables to memory addresses.

In irregular programs, this type of analysis is rendered impossible by the heavy use of pointers: the actual memory locations pointed to by a given pointer in a program may dynamically change from one run to the other for example. Pointers also obfuscate memory access patterns.

4.1.1 Stability in the symbolic dataspace

Pointers, however, do not obfuscate patterns in the symbolic dataspace which is defined as follows:

Definition The symbolic dataspace of a program is a mapping between variables and a set of textual names representing these variables. In its simplest form, the program name of a variable can be used as its textual representation in the symbolic dataspace of the program.

In the symbolic dataspace, variables are thus associated with a programmer specified textual representation whereas in the address dataspace, those same variables are associated with an arbitrary runtime value. Furthermore, in the symbolic dataspace, the association between variable and name does not change during the execution of the program whereas this is not the case in the address dataspace (pointers can be remapped for example).

The key intuition in this work is that while pointers obfuscate patterns in the address dataspace, access patterns in the symbolic dataspace are not affected.

4.1.1.1 Motivating example

Consider the sample code in Figure 16. The presence of the pointers `left` and `right`

```
struct Node {
 int data;
 Node *left, *right;
};
vector<Node*> nodes;
/* nodes is initialized to contain some Nodes */
for(int i=0, e=nodes.size(), i<e; ++i) {
 Node *n = nodes[i];
 if(n->left && n->right)
  n->data = n->left->data + n->right->data;
 else
  n->data = 0;
}
```

Figure 16: Example of a simple loop for a tree-like structure where memory access patterns are not obvious at the beginning of each iteration

makes the exact locations `n->left->data` and `n->right->data` difficult to predict. For example, the memory offset between `n->data` and `n->left->data` is not fixed between loop iterations and therefore meaningless in determining a memory access pattern.

However, in the symbolic dataspace, the access patterns are predictable and consistent: each iteration will access `n::data`, `n::left::data` and `n::right::data` where the "`::`" is used to indicate a parent-child relationship. While this knowledge does not allow for static extraction of memory access patterns, it does allow the generation of functions that can, at runtime, convert the symbolic dataspace patterns to address dataspace patterns and therefore determine, just in time, the memory access patterns of the loop.

4.1.1.2 Symbolic dataspace patterns

The stability of names in the program's symbolic dataspace makes it worthwhile to reason about access patterns in terms of names instead of memory locations. Therefore, instead of saying that a section of code accesses memory location `0xdeadbeef`, one could say that it accesses the abstract location `node::data+4` which would mean an offset of `4` within the field `data` of the variable `node`. The exact conventions used in naming variables are explained in Section 4.2.5.

Reachability Apart from the stability advantages, the symbolic dataspace notation also makes explicit a *reachability* property. Reachability from data element A to B is defined as follows:

Definition B is said to be *reachable* from A if knowledge of the address of A gives knowledge of the address of B at runtime.

In Figure 15, the address of any element `dataArray[i]` can be known from the address of `dataArray` with the addition of a fixed offset.

The symbolic dataspace notation brings this reachability property to pointers. For example, in Figure 16, the element `n->left->data` is reachable with knowledge of `n` provided the `n->left` pointer does not change. The notation `node::left::data` makes explicit the fact that the physical address of `data` is reachable from `node`, `node::left` and `node::left::data`. In other words, knowledge of, for example, the address of `node`, allows for the automatic computation of the address of `node::left::data` provided that the offsets of the various fields are known[1].

Here, the reachability is due to pointer "hopping" as opposed to offset hopping. The symbolic dataspace notation proposed supports both types of hopping.

[1] Offsets are known at compile time and therefore are easy to obtain.

Reachability is key to predictability: if data elements read or written by a section of code are all *reachable* from another common element N, then knowledge of N enables knowledge of all memory locations accessed. In Figure 16, `n` is such a common element and the memory addresses of all accessed elements can be deduced from knowledge of `n`'s address. This situation is very similar to the one shown in Figure 15 in the case of dense matrices.

Therefore, *moving pattern analysis to a program's symbolic dataspace gives irregular pointer-based programs the same analyzability as dense matrix ones.*

4.2 Symbolic dataspace memory analysis

Section 4.1 described the usefulness of describing access patterns in the symbolic dataspace. This section details a framework capable of profiling memory accesses in the symbolic dataspace. The proposed profiler is very similar to a traditional memory profiler that would indicate which memory locations are touched when except that the symbolic dataspace profiler further extracts symbolic patterns due to reachability: the profiler will therefore give all the *names* of each memory location accessed. Note that a location may have multiple names due to pointer aliasing; the proposed profiler tracks aliases and will report all possible names for a location.

The profiling runtime therefore has the following goals:

- Describe memory accesses in the symbolic dataspace.
- Construct, at the start of each section of code that is of interest to the programmer (for example transactions in STM), a function that can quickly and with good accuracy determine the memory locations that will be accessed by the section of code using reachability information. This function, equivalent to the `isDisjoint` function described in Chapter 3, can help in determining whether or not a transaction is likely to conflict with other currently running transactions.

- Provide other feedback to the programmer about access patterns in the symbolic dataspace of the program.

4.2.1 A profiling approach

The proposed framework takes a profiling approach because it only seeks to provide a function that captures the neighborhood of an operation with high probability. Obtaining a precise and accurate representation of the neighborhood is not always feasible and a statistically significant representation of the neighborhood is usually enough to determine whether or not two operations will conflict. In particular, dynamic remapping of pointers causes the mapping between names in the symbolic dataspace and program variables to change over time making determining a precise neighborhood very difficult. A static analysis, as opposed to a profiling approach, would risk getting confused by the pointer aliasing and not be able to extract any meaningful patterns.

4.2.2 Components of the profiler

The framework is divided into three parts:

C++ template wrappers C++ wrappers are provided to the programmer to associate a *name* with variables that he wants to track. Note that although this requires programmer intervention, an automated process could be devised where all variables would be wrapped and given names based on their program names. This would, however, increase the overhead of profiling as more information would have to be collected. The wrappers identify to the profiler which variables are of interest. Typically, these are the variables that are potentially accessed in transactions (using STM terminology). The C++ API also provides mechanisms to identify sections of code that are of interest to the programmer, typically the transactions themselves.

A profiler The profiler is responsible for compiling the application's source code and adding profiling annotations. When running, the application will dump a profiler file which is analyzed by the analyzer.

An analyzer The analyzer takes as input the information dumped during the application's execution and is capable of synthesizing `isDisjoint` functions among other things. The analyzer is distinct from the profiler as this limits the overhead of the profiler. The analyzer must also be capable of summarizing different executions of the same section of code (for example code that occurs in a loop). The current analyzer also provides an interactive environment where the programmer may gain additional insights on the symbolic dataspace access patterns.

4.2.3 Terminology

Throughout this chapter, the following terms will be used:

Focus area A focus area is a section of code that the programmer wishes to analyze in terms of access patterns. Typically, a focus area will be a transaction. One of the goals of the framework is to be able to determine a neighborhood function that takes as input the variables that are available at the *start* of the focus area and produces a list of memory locations that will be accessed during the execution of the focus area.

Location name A location name is a programmer-defined name associated with a physical memory location. Note that the same memory location may have multiple location names associated with it (due to aliasing). Furthermore, a given location name may refer to different memory locations at different times in the program (due to pointer remapping). The syntax of location names is defined in Section 4.2.5.

Semantic memory map The semantic memory map is the mapping between the semantic dataspace and the physical memory space. This mapping changes dynamically as the program executes: during memory allocation and deallocation as well as during pointer remapping. It will be denoted $\mathrm{SM}(i)$ where i is an instruction count.

4.2.4 Operating principle

To accurately profile memory accesses in the symbolic dataspace, it is crucial to correctly build and update the mapping $\mathrm{SM}(i)$. If this mapping is correctly maintained, the analyzer will be able to map the memory locations accessed by an instruction at instruction count i to its symbolic name.

Crucially, the names associated with each memory location are *dynamically constructed* to express the reachability property. For example, suppose a variable of type `Node` has two pointers `left` and `right`. At runtime, the `Node` will have a particular name which may not be known at compile time, suppose the name `n1`. The memory location being pointed to by `left` will be associated with the name `n1::left`. The `::` symbol represents a *parent-child* relationship and symbolizes that one can deduce the memory location of `n::left` by "hopping" through `left` from `n1`.

Operations modifying $\mathrm{SM}(i)$ are:

- Malloc/New operations create a mapping between a physical location and a name.
- Free/Delete operations remove such a mapping.
- Pointer arithmetic operations modify a mapping by associating a name with a different physical location
- Other memory operations such as `memmove` or `memcpy` also alter the mapping.

These operations are therefore monitored by the profiler and allow the building of $\mathrm{SM}(i)$ at every instruction. The profiler also monitors loads and stores and, for a load or store at instruction count i, it can determine the name corresponding to the memory location of the load or store by using $\mathrm{SM}(i)$.

Note that to minimize the profiling overhead, information is only collected on operations on the variables wrapped with the C++ API. In particular, only those loads and stores that may be relevant to the wrapped variables are profiled.

4.2.4.1 Information collected

This section describes the information that is collected for each of the operations described above. Note that for all operations, the file and line number of the operation are also collected.

Allocation The profiler cares about two types of allocations: **i)** the allocation of a pointer and **ii)** the allocation of any other data element that is not a pointer. It is important to distinguish the two because the mapping of the name to the memory location is different in each case.

In the case of the pointer, the name will be associated to the memory location that is *pointed-to* by the pointer as opposed to the memory location of the pointer itself. In all other cases, the name is associated with the memory location of the object that is allocated. Note that it is of course possible to combine both types of allocation and associate a name to the memory location of the pointer and associate a different name to the memory location being pointed to.

On allocation of a pointer, the profiler therefore collects:

- The location of the pointer as well as its size
- The memory location being pointed to as well as its size (if known)
- The name the programmer wishes to associate with this allocation

The first piece of information will be used in linking the pointer to its 'parent' element to be able to construct reachability relationships. Here, the memory location pointed-to by the pointer will be reachable from the parent containing the location of the pointer through pointer hopping. The memory location being pointed to is obviously important as this is what the name is associated with.

On allocation of any other regular object, the profiler does not collect the first piece of information as it can use the second piece of information both to determine the parent and to associate the name with the correct memory range.

Deallocation On deallocation, the profiler only needs to record the address being deallocated as this is sufficient to find which corresponding block of memory was allocated at that address and to remove the name associated with it.

Pointer arithmetic On a pointer arithmetic operation (such as adding or subtracting a constant or assigning a new value to the pointer), the name previously associated with the originally pointed-to location becomes associated with the new location being pointed to. Therefore, on a pointer arithmetic operation, the profiler collects:

- The address of the pointer
- The original address being pointed to
- The new address being pointed to

The first two items allow the positive identification of the name that is being changed and the last item allows the correct remapping of the pointer.

Loads and stores The address and size of the load or store are collected by the profiler. The semantic memory map constructed by the monitoring of the previously

covered operations allows the address of the load or store to be associated with the names for that location thus enabling memory profiling at a semantic level.

4.2.5 Naming conventions

Section 4.2.4.1 described the information that is collected by the profiler. The names collected in this process are only partial names and do not reflect the parent-child relationships that are crucial in establishing reachability. It is the job of the analyzer to reconstruct the full name of each memory location by determining the *parent* of each memory location. The parent of a memory location is a named enclosing memory location: this is the traditional "offset" child of dense matrix operations. However, to support pointer hopping, the parent of a memory location being pointed-to is the parent of the pointer.

The notation used uses the '`::`' symbol to separate two names, the latter being a *child* reachable from the former (the *parent*) using a single hop (pointer or offset). For example the name `n::data` indicates that `data` is a direct child of `n`. The name does not reflect whether or not `data` is an element pointed to through a pointer contained in `n` or a member element of `n` as it is irrelevant from the point of view of reachability. The analyzer does however maintain this information as it is relevant in constructing a function to determine the memory location of `n::data` from that of `n`.

While the naming convention expresses in a straightforward and composable way the reachability property required to analyze pointer-based programs, specific issues need to be addressed:

- The name of the parent may not be known at compile time. To solve this, the name of the parent can be dynamically discovered at runtime. This technique also enables greater composability. For example, in a classic binary tree, each node's name will reflect the exact path followed within the tree from the root node.

- The number of children may be dynamic and therefore make the static naming of children impossible. To solve this, the name of children may be auto-generated by the analyzer.
- The utilization of library code which the programmer cannot or does not want to annotate means that the name of a data element may be incomplete.
- Multiple names may point to the same memory location due to pointer aliasing.

The following sections address these problems.

4.2.5.1 Dynamic determination of parent-child relationship

Consider for example the `Node` data-structure in Figure 16. The full names of the `left` and `right` pointers should be of the form `<parent>::left` and `<parent>::right`. However, statically, the name of the parent is not known. It can, however, be determined at runtime by determining the name of the region of memory in which the allocation occurs.

Allocation of an "offset" child An "offset" child is an element related to its parent because a fixed offset relationship exists between its address and that of its parent. This is typically a member field in C++ or an element in a struct or vector in C. This type of child is the typical relationship that is tracked for dense-matrix operations. Determining the parent for such a child is simply a matter of determining the name associated with the memory region in which the child is allocated.

Allocation of a "pointer" child A "pointer" child is an element related to its parent because a pointer links the parent to the child. In this case, note that the physical memory location of the child is not related to that of the parent in any discernible way (in fact, depending on the allocation mechanism, it may change between runs). However, determining the parent is still straightforward as the profiler records,

for a pointer, both the address of the pointer as well as the address of the object being pointed-to. The pointer itself is an "offset" child of the parent and this relationship makes it easy to link "pointer" children to their parents.

Figure 19 illustrates the concepts of "offset" child as well as "pointer" child.

4.2.5.2 Dynamic determination of a child's name

Consider for example the sample code in Figure 17 where the macro `TRACKED(object, name)` represents the wrapping API required to associate `name` with `object` and `TRACKED_PTR(ptr, name)` represents the same API for pointers. In this example, the

```
struct Node {
  vector<TRACKED_PTR(Node*, "::")> neighbors;
};
TRACKED(Node n, "n");
```

Figure 17: Illustrative example for the use of virtual children. `TRACKED_PTR` indicates that `neighbors` contains pointers to `Node` objects that are tracked by the framework. The '`::`' name indicates that the name will be auto-generated by the analyzer. `TRACKED` indicates that the creation of `Node n` creates a mapping between the memory space occupied by `n` and the name "`n`".

elements of `neighbors` are children of `n`[2]. However, their number is unknown and may be dynamically changed at runtime. Their names cannot therefore be statically determined and must be generated at runtime. The analyzer does this by simply assigning a sequential number to each allocated child. In this particular case, the first child would be named `n::1`, the second `n::2` and so on. Note that the assigned number may or may not correspond to valid indices in `neighbors`.

4.2.5.3 Virtual children

The determination of a parent-child relationship as described in Section 4.2.5.1 relies on the fact that a relationship between the physical memory of the parent and the child (or the pointer to the child) exists. However, in certain cases, although such a

[2]Technically, they are virtual children, see Section 4.2.5.3.

relationship may exist, it may be not be accessible to the programmer. Consider again the example in Figure 17. The elements of `neighbors` are logically children of their parent `Node` but do not have either a fixed offset or pointer relationship that is known to the programmer. In this specific case, the C++ vector is actually implemented with the use of a pointer that points to the elements contained in `neighbors` but this is not accessible to the programmer.

The framework therefore allows the programmer to identify such containing data-structures (vectors, lists, sets among others) which will cause any children of these data-structures to be associated in a parent-child relationship with the parent of the containers. This is referred to as a *virtual* parent-child relationship.

4.2.5.4 Namespace aliasing

Due to pointer aliasing, it is possible for the same physical memory location to be identified by multiple names. This is denoted using the symbol '—'. For example, `(n2|n::left)::data` indicates that `n::left` and `n2` alias each other. The complete expansion of the names gives all possible semantic 'paths' to a specific memory location. The term *depth* will refer to the number of '`::`' in the path and corresponds to the number of 'hops' required to reach the address of the named element from the address of the top-most parent.

4.2.5.5 Naming grammar

To summarize, the grammar for a name is as follows (EBNF grammar):

⟨*full_name*⟩ ::= [⟨*parent_name*⟩, '::'], ⟨*terminal_name*⟩;

⟨*parent_name*⟩ ::= ([⟨*parent_name*⟩, '::'], ⟨*atom_name*⟩) | ('(', ⟨*full_name*⟩, '|', ⟨*full_name*⟩, ')');

⟨*atom_name*⟩ ::= '_' | ⟨*terminal_name*⟩;

⟨*terminal_name*⟩ ::= (⟨*alpha_chars*⟩ , ⟨*all_chars*⟩) | (⟨*digit_chars*⟩ ,

⟨*digit_chars*⟩);

⟨*all_chars*⟩ ::= ⟨*digit_chars*⟩ | ⟨*alpha_chars*⟩ ;

⟨*digit_chars*⟩ ::= ? 0 to 9 ? ;

⟨*alpha_chars*⟩ ::= ? A to Z and a to z ? ;

4.2.6 Relationship between symbolic dataspace and address dataspace

Consider the code segment in Figure 18. This code segment, representative of data

```
1 struct Node {
  TRACKED(int data, ``::data'');
  TRACKED_PTR(Node *left, ``::left'');
  TRACKED_PTR(Node *right, ``::right'');
  };
6
  TRACKED(Node n, ``n'');
  TRACKED(Node n2, ``n2'');
  n->left = &n2;
```

Figure 18: Sample code segment illustrating the used of "offset" children and "pointer" children as well as aliasing.

structures used in tree-based algorithms, illustrates the use of "offset" children (`data`) as well as "pointer" children (`left` and `right`). It also illustrates the use of aliasing between `n::left` and `n2`.

The semantic memory map constructed by the analyzer at the end of this code segment is shown in Figure 19. The association of the name `n` with the memory region $[0x10; 0x24]$ is caused by Line 7 and the association of `n2` with $[0xA0, 0xB4]$ by Line 8 [3]. Line 9 creates the relationship between `n::left` and `n2`. One can see that all the information required to build such a memory map is captured by the profiler (see Section 4.2.4.1) and that the memory map built fully captures reachability information.

[3] Memory addresses were chosen arbitrarily.

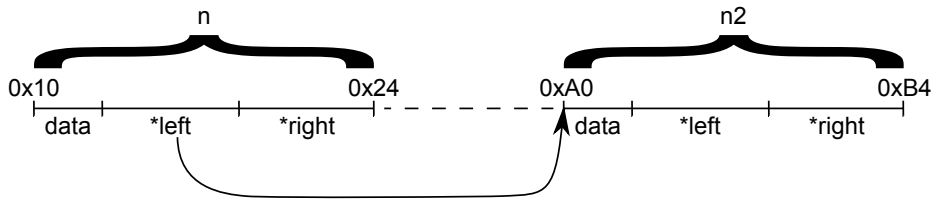


Figure 19: Memory map constructed by the analyzer at Line 9 of the code segment shown in Figure 18

4.3 Implementation

This section describes the implementation details of the framework detailing each of the three parts separately: the C++ API, the profiler and the analyzer.

A C++ API The API is used by the programmer to annotate the variables in the program that he cares about. The same annotation also allows the programmer to associate a name to the variable. Finally, the API allows the programmer to define the focus areas (see Section 4.2.3).

A compiler and profiler The compiler takes the annotated source code and produces an instrumented version of the application that, when run, will profile itself and dump a profiler file (referred to as `profiler.dump`) which contains information concerning allocation, deallocation, pointer remapping as well as the loads and stores of the variables that were annotated by the programmer.

An analyzer The analyzer takes the raw `profiler.dump` and transforms it into a human-readable one. In particular, it outputs all loads and stores in the name space of the program (as opposed to its address space) and ranks the variables at the start of each focus area in terms of their predictability power (in other words, the ability to *reach* the other loads and stores that can *reach* (*predict*) the maximum number of loads and stores). The analyzer also provides an interactive command line interface that allows the programmer to further analyze the access patterns in name space.

4.3.1 C++ API

The main role of the C++ API is two-fold: **i)** provide a means to identify focus areas and associate a name with them, and **ii)** provide a mapping between program variables (address dataspace) and variable names (symbolic dataspace) However, the API also fulfills a secondary role of allowing the profiler to selectively profile the loads and stores to reduce the profiling overhead as well as the size of `profiler.dump` which is the file that the profiler will write to during the execution of the program.

4.3.1.1 Identifying focus areas

A focus area is any scoped block of code. It can be a function or the code inside a `for` loop. To identify a block of code as a focus area, the programmer uses the macro `IDENTIFY_BLOCK(<name>)`. This macro will cause two lines to appear in the `profiler.dump` file, one recording the instruction count at the start of the block and one recording the instruction count at the end of the block.

Note that if a block of code is repeatedly entered, a unique name for each instance is created by appending an integer to `<name>`.

4.3.1.2 Identifying variables

Variables of interest are those that are read or written to during the execution of a transaction; in other words, they are those on which access conflicts may arise. To profile in the symbolic dataspace, these variables need to be given a name.

C++ templates are utilized to associate names with variables. The templates wrap the original variable in such a way that they behave similarly to the original variable but monitor access to the variable. When the wrapped variables are accessed, a flag is set indicating to the profiler that loads and stores should be tracked. The loads and stores tracked will be a superset of those to the wrapped variables. The wrappers also allow for controlled allocation and deallocation of the variables thereby ensuring the the names are associated with the correct memory regions.

To differentiate between pointer variables and non-pointer variables, the framework offers two wrappers `Tagged<T>` wraps "offset" variables while `PtrToTagged<T>` wraps "pointer" variables. Note that taking the address of a `Tagged<T>` object will produce a `PtrToTagged<T>` object thereby maintaining any named association with the memory location referenced by the `Tagged<T>` variable.

The framework also provides a `TaggedContainer<T>` wrapper which provides support for the virtual parent-child relationship. When a `TaggedContainer<T>` object is accessed, the address of its parent is pushed on a runtime stack and future tracked allocations will be tagged as virtual. The analyzer will use the information pushed on the stack to recreate the correct parent-child relationship.

Automatic variable tagging While the current framework requires the programmer to identify and name the variables that are of interest, in most cases, it is possible to identify these variables automatically using a source to source translation layer.

The variables of interest are the ones that are accessed within the focus areas identified by the programmer. A source to source translator could identify the names of the variables accessed in this region and automatically change their types using the API previously discussed.

4.3.2 Profiling pass

The profiling pass is implemented as a LLVM [52] pass which annotates the original source code. In particular, it adds the following annotations:

- Updates a global dynamic instruction counter. The instruction count is used instead of time to indicate the 'when' of a load/store event. Since the framework is only interested in determining patterns and not execution times, using the dynamic instruction count as a measure of execution progress is not problematic and will provide more stable and accurate results because it ignores any

architectural effects such as cache misses which would be visible in an approach that measured absolute time.

- Every load and store is annotated. However, to reduce overhead, the annotation consists of an initial check for a flag. If set, the actual instrumentation is called, otherwise, the program continues. This greatly reduces overheads and allows the output file to be considerably shorter than it would have been if all loads and stores were profiled.

- Certain memory operations and intrinsic are also annotated. In particular, `memmove` and `memcpy` are annotated as they potentially change the memory map.

Note that `malloc` and `free` are not annotated by the profiler, instead, the C++ wrappers directly print out in the `profiler.dump` file on allocation and deallocation of wrapped variables. The same is also true for any pointer arithmetic on named variables.

4.3.3 Analyzer

The analyzer is a Python script that takes as input the `profiler.dump` file and **i)** builds SM (i), **ii)** converts the memory locations accessed by loads and stores to names and **iii)** determines for each focus area the variable that best predicts future accesses. An interactive mode is also available which allows the programmer to determine the LocationName from which the memory loads and stores can be predicted. In other words, the analyzer gives the `isDisjoint` function: at the start of each focus area, the analyzer will print out the LocationNames which allow the prediction of the memory accesses within the focus area.

4.4 Experimental validation

The C++ API and LLVM pass are release quality products, however the analyzer, although it is functionally complete, does not yet scale to large applications. This is mainly due to the fact that for applications that have many pointers (for example a dense graph), a single node has possibly hundreds of names as the number of paths terminating in that node can be very large. For this reason, the current analyzer can only deal with smaller input sizes. This does not detract from its usefulness as the patterns extracted for smaller input sets are likely to also be present in larger input sets.

4.4.1 Experimental setup

The graph coloring example, previously presented in Section 3.2, was used to demonstrate the applicability of the framework. Note that this benchmark is trivial and nothing new will be learned about its access patterns. However, this enables the validation of the tool: the output of the tool will match what is expected, namely that each invocation of `processNode`, the function that colors each node, reads the neighbors of the `input_node` and writes to the `color` field.

The actual code of `processNode` is given in Figure 20 Several points are worth mentioning. Line 15 defines the name `input_node` and associates it with the current node that will be processed. Note that the exact aliasing will change over time (each time a new node is processed). However, having a single name for the input allows the programmer to make a lot more sense of the summarized access patterns across multiple instances of `processNode`. Line 17 shows the API to identify a section of code the programmer is interested in. One can easily see that the main computation looks at all the neighbors of the `input_node` and writes a suitable color to `input_node::color`.

```
void processNode(_rec::PtrToTagged<Node*>& node,
    _rec::PtrToTagged<Graph*>& graph) {
  vector<bool> used(graph->nodes->size(), false);
  for (unsigned int j = 0; j < node->adjacent_to->size(); ++j) {
    int color = *(node->adjacent_to[j]->color);
    if (color != -1)
      used[color] = true;
  }
  unsigned int smallest_color = 0;
  while (smallest_color < graph->nodes->size() &&
      used[smallest_color] == true)
    ++smallest_color;
  node->color = smallest_color;
}

void processNode(int node_number, _rec::PtrToTagged<Graph*>&
    graph) {
  _rec::PtrToTagged<Node*> node(getNode(node_number, graph),
      "input_node", false);

  {
    IDENTIFY_BLOCK("processNode")
    processNode(node, graph);
  }
}
```

Figure 20: Algorithm used in the experimental setup: Greedy graph coloring. The code shown is the actual code using the C++ API provided. The APIs are in the `_rec` namespace.

4.4.2 Note on overheads

The overheads of the framework described are very low as far as the profiler is concerned. In particular, the profiler does not do any complex analysis of the namespace, restricting itself to simply printing out the memory addresses loaded from and stored to. The profiler also tracks certain allocations and deallocations. These operations are relatively rare and therefore cause little overhead. For the loads and stores, the profiler smartly only considers the loads and stores that may be of interest (in other words, that may refer to a named memory location) and ignores the rest (mostly). This greatly reduces the size of the `profiler.dump` file and also the overhead of

profiling.

The analyzer on the other hand is an extremely compute intensive piece of code and is currently very memory hungry. However, this analysis is offline and will most likely only be done once.

4.4.3 Results

After running the instrumented code on an input with 10 nodes and 2 edges on average for each node, a `profiler.dump` file was produced which details the loads and stores in terms of memory addresses. The analyzer then interpreted the file and translated it into the symbolic dataspace and entered an interactive session. The analysis of the `processNode` yielded the summarized information shown in Figure 21. The first part of the output displays the association between an internal analyzer ID

```
——— ID to LocationName correspondance ———
0 -> main_graph
1 -> main_graph::nodes
13696 -> input_node

——— Formula W = Sum(1) (count = 24.60 (stdDev = 9.20), total W
    =  24.60 (stdDev = 9.20)) ———
Group (     [0]) -> 24.60 stdDev= 9.20 (100.00 %)
Group (     [1]) -> 24.60 stdDev= 9.20 (100.00 %)
Group ([13696]) -> 19.00 stdDev= 8.58 (77.24 %)
```

Figure 21: Results showing the summarized information over ten invocations of `processNode`.

and the programmer assigned name. The ID is used in the second part which shows the number of accesses that can be accessed from the name. One can note that all accesses are reachable from both `main_graph` and `main_graph::nodes` which makes sense because everything is contained in those elements. Furthermore, `input_node` predicts on average 77.24% of all accesses. This is because it does not predict accesses such as those on Line 2 in Figure 20. This is consistent with what would be expected. The analyzer further knows the exact paths to determine the memory locations of

each of those accessed elements and can therefore aid the programmer in determining that he should look at `input_node`'s neighbors and `color` field.

4.4.3.1 Playing with event weights

In the previous results, each memory event (either a load or a store) was attributed the same weight (1). Basically, the analyzer was just counting the number of events. However, more complicated and interesting ways of counting events are possible. For example, Figure 22 shows the results if only write events are counted and the weight for each write event is divided by the number of hops required to reach the actual location being written to and the LocationName from which it is reachable. In other words, the weight will only be 1 for a particular event if and only if it is a write event to that specific LocationName (depth of 1). The results show that only

```
1 ---- ID to LocationName correspondance ----
  14422 -> input_node::color
  15383 ->
      main_graph::nodes::_::16::neighbors::_::3::neighbors::_::10::neighbors::_:

  ---- Formula W = Sum(w/(d+1)) (count = 1.00 (stdDev = 0.00),
      total W =  24.60 (stdDev = 9.20)) ----
6 Group ([14422]) -> 1.00 stdDev= 0.00 (4.07 %)
  Group ([15383]) -> 0.10 stdDev= 0.30 (0.41 %)
```

Figure 22: Results showing the summarized information over ten invocations of `processNode` with only write events being counted and weighted inversely by the number of hops required to reach the data element of the event.

`input_node::color` has a consistent weight of 1 and no other LocationName which is consistent with what is expected.

4.5 Conclusion

This chapter demonstrated the possibility of analyzing memory access patterns in the *symbolic* dataspace of a program. This chapter demonstrated that the extraction of these patterns could be used in determining the memory footprint of an operation

thereby enabling a runtime such as the one described in Chapter 3 to intelligently schedule concurrently running transactions.

The work described in this chapter lays the groundwork for a more ambitious framework that is future work which will be able to predict footprints in a context sensitive manner and also tackle larger and more complex applications than the one we presented in Section 4.4.

4.5.1 Thesis discussion

Chapter 3 demonstrated that semantic knowledge from the programmer could usefully be exploited to improve parallelization. This chapter goes further and shows how semantic information is sometimes already present in a program but is not currently being exploited because it is obfuscated. This chapter detailed how changing the level of abstraction, from the address dataspace to the symbolic dataspace, can reveal additional information that can be put to use in a framework such as the one described in Chapter 3. This furthers the thesis statement that algorithmic properties can be used to improve parallelization. Indeed, the change from the address dataspace to the symbolic dataspace exposes certain properties of the algorithm; the profiling tool automatically extracts these properties.

Profiling in the symbolic dataspace also has other uses that further the thesis statement. In particular, although this chapter details the use of the symbolic dataspace to determine the data-footprint of a computation, the *access patterns* extracted in the symbolic dataspace can also be used to generate an *application signature*. In particular, this could be useful in the N-Way framework to evaluate the diversity present among the ways: ways that have similar access patterns could be considered similar and therefore exhibiting low diversity. Working in the symbolic dataspace again exposes the algorithmic properties of each of the ways: although the exact memory locations touched by each way may be different, the application (or way) signature

would be comparing access patterns in the symbolic dataspace which means that "inputs" can be mapped (given the same name) for example. This information could be used in informing the learning and culling algorithms:

- The learning algorithm could bias its selection towards the more diverse ways.
- The culling algorithm could, instead of culling ways making little progress, cull ways that are very similar to each other as the benefits of N-Way are directly related to the amount of diversity expressed.

To integrate with N-Way, a realtime framework would be preferable and is the object of future work. The current analysis performed by the analyzer is too intense to execute at runtime but it could be made more efficient by making simplifying assumptions on the access pattern or trying only to detect broad differences between ways instead of constructing an accurate signature.

CHAPTER V

QUALITY DRIVEN COMPUTING THROUGH VARIABLE SEMANTICS

The N-way model presented in Chapter 2 proposes a solution to the sequential bottleneck problem described in Section 1.2.2 by launching multiple competing ways to gain speedup and QoR improvements. N-way exploits algorithmic diversity in certain problems to do this. This chapter exploits another characteristic of certain problems, *variable semantics*, to launch *collaborating* ways to improve the quality of result.

5.1 Shifting application characteristics

As motivated in Chapter 1, hardware is becoming more and more parallel. However, concurrently to this shift, applications are also undergoing an evolution from traditional HPC applications to more consumer-driven ones. Computers have moved from being used solely for office work to hosting games and multimedia applications. More specifically they now support what are called "immersive environments" which seek to provide a more engrossing experience to the user. Games are a prime example of this as they seek to immerse the user in a virtual world. Desktop environments and browsers are also trying to be more immersive to offer a seamless and intuitive experience to the user.

The immersion present in these newer applications exposes two important characteristics that most classical applications did not have: **i)** variable semantics and **ii)** a responsiveness requirement.

Variable semantics is defined as the fact that for a given problem, there can be multiple *correct* solutions, some potentially better than others but all solving the

same problem. Examples of this were given in Section 1.4.3

The other characteristic strongly displayed by immersive applications is *responsiveness* which is defined as a quick response to user input and providing him/her with a smooth experience. In the gaming domain for example, this means that the screen must be refreshed at a speed of more than 30 frames per second (so that the human eye does not perceive the refresh rate) and that user inputs (such as commands to the game hero) must be addressed quickly (ie: if the user wants to turn, the turn should happen instantaneously).

This thesis focuses on games as an example of applications that exhibit these characteristics.

5.1.1 Parallel programming in games

Currently, parallel programming in games relies on the traditional techniques of breaking up the computation and/or data into distinct parts that can be processed in parallel. These approaches are complex and only lead to somewhat incremental improvements [74, 22, 19]. Moreover, since games are often written in C/C++, they involve the use of data structures that rely heavily on pointers which complicates parallelization due to difficult to analyze data-sharing patterns. Finally, these approaches do not take advantage of the specific characteristics identified in games: variable semantics and a responsiveness constraint.

This thesis proposes an alternate approach where the programmer does not explicitly need to focus on parallelizing the game itself but rather indicates to a runtime how it can utilize the parallel cores available to opportunistically launch *quality improvement* tasks. This approach therefore takes advantage of the flexibility allowed by the variable semantics This quality-driven approach attempts to answer the following goals:

- Maximize the quality of game computation by leveraging as many parallel processing cores as possible.
- Respect the responsiveness constraint by always providing an acceptable answer within the time frame available.
- Free the programmer from the difficult task of explicitly parallelizing game code.
- Allow the programmer to *design once* for a wide range of platforms. Indeed, since the program will dynamically morph depending on resource availability, it will be able to adapt to a wider range of platforms. Currently, games need to be redesigned for newer platforms.

5.2 A quality based approach

In a regular single-threaded program, program flow is very well defined and the programmer can easily see and trace what the program is going to execute next. In a multi-threaded application, many such threads exist and execute in parallel in a manner known to the programmer. In the quality-driven approach, the actual program executed will be dynamically determined and changed by a runtime depending on **i)** the quality requirements expressed by the programmer and **ii)** the available processing resources and time to compute a result.

It is important to note that this does *not* mean that the application will be unpredictable. The quality framework allows the programmer to define what quality is acceptable and as such, certain minimum requirements will always be met. If resources permit, however, the quality produced will be much higher than the minimum required.

This section describes the notion of *quality* as well as details the execution model briefly described above.

5.2.1 Notion of quality

Key to the quality-driven approach is the notion of quality of a computed result. This notion stems directly from the variable semantics characteristic in applications. As established earlier, multiple results are acceptable in games; therefore, a way to distinguish these results and rank them needs to be established: the *quality* of the result.

The notion of quality is difficult to define in general as it is largely program dependent. As far as the approach is concerned, quality only needs to be an attribute than can be used to determine which result is *better* than another: it is thus domain agnostic

Quality parameters Quality is defined differently for different objects in a game. For example, for a bot (a simulated player controlled by AI), quality will be defined as the level of intelligence it exhibits, for a rendered scene by the level of detail in it and for a physics simulation by the accuracy of the simulation. All of these aspects of quality define quality *parameters* which can be viewed as different dimensions of quality. Quality parameters must be defined at the start of a program and their number must remain constant throughout the execution of the program. It is supposed that there are N_q quality parameters in the program. Identifiers for quality parameters range from 1 to N_q.

Quality value A quality value is defined as a tuple, of size N_q. Each element of the tuple is either an integer or the special value 'NaN'. Element i of the tuple corresponds to the value for the quality parameter i. The value 'NaN' indicates that the quality parameter does not apply to the object. For example, suppose there are three quality parameters indicating the level of AI intelligence, the level of rendering detail and the accuracy in physics simulation. A bot object would have a quality value of the

form (x, NaN, NaN) while a graphical object to be rendered would have a value of the form (NaN, y, NaN).

The operations on quality values are:

- Change a parameter value: each integer parameter value can be incremented or decremented.

- Compare quality values: quality values that consider the same quality parameters (in other words, that have 'NaN' in the same positions) can be compared. To compare two quality values, a weighted element by element difference of the non-NaN elements is computed. For example, to compare $V_1 = \left(a_1, \ldots, a_{N_q}\right)$ and $V_2 = \left(b_1, \ldots, b_{N_q}\right)$, $R = \sum_{i=1}^{N_q} \left(w_i \left(a_i - b_i\right)\right)$ would be computed where the w_is are programmer defined weights (which default to $1/N_q$) meeting the constraint $\sum_{i=1}^{N_q} w_i = 1$. A positive R would indicate $V_1 \geq V_2$.

Use of quality As an example, consider a particle simulation system where the next position of a particle is determined by the position of its neighbors and a force field (wind, gravity, etc.). In such a system, two quality parameters could be introduced:

- The distance of the furthest neighbor taken into account to calculate the position;

- A boolean indicating if the force field was taken into account.

A particle position object would be associated with a quality value of the form (x, b) where x would indicate the distance and b whether or not the force field was taken into account. A quality value of $(10, true)$ would be better than one of $(5, false)$ for example as the simulation would be more accurate.

Conceptual definition of quality Conceptually, the quality attribute of an object represents the types of modifications that are being tracked on the object. If the

object is modified, its attached quality attribute should reflect that change. Quality parameters define the types of modifications and some examples are:

- **Accuracy level** For example, if a program is calculating the Taylor series expansion, a quality parameter could track the number of terms that were used to calculate the expansion;

- **Precision level** Current languages provide `float` and `double` to allow computations at various levels of precision. The precision of a value could also be a quality parameter and used to estimate the error on a result for example;

- **Algorithm Alternative** A quality parameter could indicate which of a set of possible computation has been applied to a data element. In a game for example, such a parameter could be used to track which decision method was used in an AI algorithm.

5.2.2 Program flow

To explain the concepts of a quality-driven program flow, consider a single-threaded application. Note that this approach will effectively make the application multi-threaded but the programmer only needs to write a single-threaded application.

A quality-driven execution is a regular single-threaded execution annotated with extra quality requirements on certain data objects. The key idea is that the programmer sets quality requirements on data elements and the runtime then applies, in parallel, transformations to the data to obtain at least the required quality. Note that the requirements are *minimum* requirements but the runtime will seek to produce the best quality result depending on the resources available. This type of execution requires two types of information from the programmer:

- The specification of quality requirements.

- The specification of *Transformers* that take data from a certain quality and modify it in a way that enhances its quality. Transformers can be viewed as pure functions which improve the quality of their inputs. Transformers are dynamically and automatically combined by the runtime to meet the requirements specified by the programmer.

Unique approach The approach proposed is novel in the sense that the programmer does not focus on *how* a result is obtained but rather on *what* result is desired. The runtime will dynamically compose different Transformers in parallel to obtain such a result. Note that the possibility of combining various Transformers directly stems from the variable semantics characteristic of games. The main thread of execution (written by the programmer) is only responsible for specifying quality requirements and the computation required to compute the "non-variable" values (those to which variable semantics do not apply). The main thread therefore ensures that the program meets the minimum standard required of it. While the main thread is single-threaded, the running of Transformers in parallel will effectively make the application multi-threaded.

5.2.2.1 Specifying quality requirements

Quality requirements can be specified using the following types of calls:

- **Quality requirement** The programmer can require a specific data element to be of a requested quality at a specific point in the program. Note that given that multiple results may be acceptable, this requirement can involve **i)** a minimum acceptable requirement, **ii)** a preferred requirement or **iii)** a trade-off requirement where the programmer is willing to wait for some time for a better result. To maximize the possibilities of getting a better result, it is best to make this requirement known to the runtime before it is actually needed (in a way

similar to *futures* [43] where a computation for a value is non blocking until it is actually used).

- **Queries** The programmer can query the runtime as to the current state of computations, the availability of results for data elements, etc. This information can be used by the programmer to check how the runtime is handling the work and debug any issues that arise.

The specific API for these calls will be given in Section 5.3.

5.2.2.2 Specifying Transformers

Transformers are simply procedures that operate on data elements with a quality attribute and transform them to produce the same data element but with a different level of quality. One can view Transformers as offloaded pure functions. Indeed, in the quality framework, the runtime will *copy* any input required by the Transformer similar to when data is copied to a GPU for processing. Note that since the computation occurs in parallel to the main program, the inputs may be changed by the main program while the Transformer is processing. In this case, the computation of the Transformer is invalidated and it starts over in a way similar to the rollbacks used in STM systems. However, since Transformers live in an isolated environment, the cost of a rollback is simply the cost of the computation that has already occurred.

5.2.2.3 Hierarchical threads

In the quality model, a "main thread" instructs the runtime of certain quality requirements. The computations launched by the runtime as a result of these instructions operate in a closed environment where all data is copied over to them (there is no sharing of data to prevent synchronization issues). Thus, each Transformer thread can also be viewed as a "main thread" operating in a new environment. The model can thus be extended to have hierarchical Transformer launches.

5.2.3 Summary

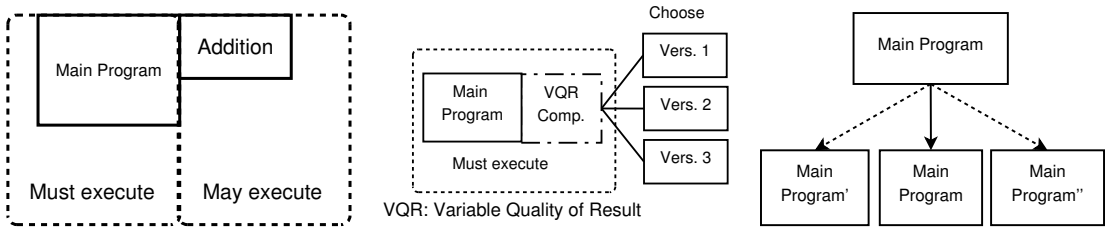
To summarize, the quality model introduces a new program flow based on quality requirements. A main thread instructs the runtime as to what it requires in terms of quality of data elements and the runtime will dynamically launch the best possible computation threads to satisfy these requirements. Each new computational thread can also make additional quality requirements.

5.3 Use scenarios and API

This section presents use scenarios for the approach as well as the API to support them. It also describes the innards of the runtime.

5.3.1 Extensible program semantics

The key concept behind the quality approach is extensible program semantics. The runtime's role is to provide the programmer with the possibility of adding, improving or morphing computations that are taking place. The exact nature of the transformations that occur are controlled by the programmer supplied quality attributes.



(a) Addition of a computation to the main program

(b) Specification of multiple execution paths for a given function

(c) Morphing of the program into an extended one

Figure 23: Extending a program's semantics

5.3.1.1 Addition

Addition is the most straightforward concept. The programmer defines an optional computation that is not absolutely required (for example, adding shadows or complex

lighting effects in a game). While these effects are visually impressive, they are not required and may unacceptably degrade performance. The use (or not) of these effects would be tracked by a quality parameter; they will only be run if sufficient resources and time are available. This is illustrated in Figure 23(a).

5.3.1.2 Revision and refinement of a result

The framework centers around the notion of quality where each data object can be associated with a quality attribute. Transformers change an object of one quality to the same object but of better quality by using certain inputs. However, when inputs change, it is possible to "revise" a calculation in progress instead of purely canceling it. An example of this is given in Section 5.3.3.2.

Refinement means that a Transformer can use a previously calculated result by another Transformer and bypass some of its computations. For example, in a program calculating Taylor expansion terms, if Transformer `A` has calculated the first 10 terms of the expansion, if Transformer `B` wants to calculate 20 terms of the expansion, it should not have to recalculate the first 10 terms.

In both cases, a previously computed result (or partial result) is reused to compute the new result thereby eliminating redundant computation. The runtime provides support for this.

5.3.1.3 Morphing

Previous concepts added small pieces of computation locally without significantly changing the overall flow of the program. In the scenario of program morphing however, a program is allowed to morph into a more resource intensive program performing a similar task. Figure 23(c) illustrates this idea.

In the MPEG encoding algorithm, for example, a task that started out coding an I frame could morph into coding a P or B frame provided enough time and resources are available. The morphing will require more resources for a longer period of time

and thus, mispredicting a program morphing can be expensive. However, it does allow interesting programming possibilities especially in soft real-time systems since deadlines are not hard.

5.3.2 API

For a program to use the quality infrastructure, two steps are required. In a first phase, the programmer must inform the runtime of all the Transformers (ie: the possibilities available to it to improve quality for a given class of objects). The programmer must also define the quality parameters that will be relevant to him and inform the runtime of them. This is the registration phase. In a second phase, the programmer will make use of the runtime by informing it of his quality requests as described in Section 5.2.2.

5.3.2.1 Registration phase

During the registration phase, the programmer must specify Transformer objects and register them with DataWithQuality objects. DataWithQuality objects are also registered with the runtime to enable the runtime to uniquely identify them.

`Transformer` objects A Transformer is defined in Figure 24. Note that all code snippets are simplified to make them more legible. A Transformer is a combination

```
class Transformer {
    /* ... */
    Transformer (void (func)(BaseType *curValue, QualityVector
       *curQuality, const UserInput<BaseType, InputType>
       *input));

    QualityVector getQualityModification (QualityVector
       *curQuality);

    time_t getTimeEstimate();
};
```

Figure 24: Definition of a Quality Transformer

of three functions:

- A **work function** as defined above as the `func` argument to the constructor. The work function will take the current value for an object, its current quality and other input data and produce the same object at a different quality level.
- A **quality modification function** which estimates how the Transformer is going to modify a data object in terms of its quality.
- A **cost estimator function** which estimates the time cost of executing the Transformer.

All three functions have to be defined by the programmer. This may seem difficult for the latter two functions but they are merely used as approximate indicators by the runtime which uses them to determine the best Transformer to use to meet the quality requirements while still meeting soft deadlines (responsiveness).

`DataWithQuality` objects A DataWithQuality object wraps around an arbitrary user-defined object and adds a notion of quality to it. A DataWithQuality instance will contain multiple values for the wrapped object, all with different levels of quality. Figure 25 shows the important aspect of this object. A DataWithQuality object thus contains the different Transformer objects that apply to it to indicate the operations that can be executed on it. It also contains a set of values (contained in `values`) which contains all the different values, at varying degrees of quality, that have been calculated for the wrapped object.

Note that DataWithQuality objects are globals that the runtime uses to keep track of all the quality-tagged objects. The Transformer threads operate on DataWithQualityVariable objects which are thread-safe instances of DataWithQuality objects.

```
class DataWithQuality {
    DataWithQuality(BaseType *toWrap);

    static TransformerId setTransformer(Transformer<BaseType,
       InputType>* transformer);

    static void addQualityType(QualityType type);
    protected:
        std::vector<DataQualityPair<BaseType> > values;
        DataWithQualityId instanceId;

        BaseType* getResultForQuality(QualityVector *quality);

        BaseType* getBestResultForQuality(QualityVector
           *quality);

        BaseType* getBestPossible(QualityVector *quality);
};
```

Figure 25: Definition of DataWithQuality

5.3.2.2 Runtime API

The runtime API has been kept very simple and only the smallest number of directives that would allow the greatest expressiveness are allowed. This section only describes the methods related to specifying quality requirements, and not the query methods which have largely a debugging role. The important functions are described in Figure 26. The calls closely match the different quality requirements that a programmer can make as described in Section 5.2.2.1. Each call takes a DataWithQualityVariable object that will be modified (except in the case of a future quality request) to contain the new value as computed by the Transformer objects associated with the type passed. All calls (except the future quality request) are blocking although some may block for longer than others. The `requireQuality` call will block until a result of sufficient quality has been calculated. Other calls will block for much less time (the `preferQuality` call will block for a very short time as it only returns values that are currently available).

```
class Runtime {
    void requireQuality( DataWithQualityVariable<BaseType,
       InputType> *variable, QualityVector *reqQuality);

    void preferQuality( DataWithQualityVariable<BaseType,
       InputType> *variable, QualityVector *prefQuality);

    void tradeoffQuality( DataWithQualityVariable<BaseType,
       InputType> *variable, QualityVector *reqQuality, unsigned
       int waitTime);

    void futureQuality( const DataWithQualityVariable<BaseType,
       InputType> *variable, QualityVector reqQuality, unsigned
       int availTime=0);
};
```

Figure 26: Principle API calls for the quality-driven runtime

5.3.3 Runtime implementation

This section describes some of the mechanisms used by the runtime to implement the programming styles described in Section 5.3.1.

5.3.3.1 Quality aware runtime

When the runtime receives a quality request from a thread in the program it will try to satisfy it as quickly as possible. The basic algorithm is given in Algorithm 2. The algorithm changes slightly depending on the type of request the runtime receives:

- For a **strict** quality requirement, the full algorithm will be used.
- For a **prefer** quality requirement, only results currently available will be used.
- For a **trade-off** quality requirement, the runtime will use the full algorithm but abort it if it goes over the time given to it by the programmer.
- For a **future** quality requirement, the full algorithm will be used but nothing will be returned to the call-point of the future in the program. At the future's wait-point (say `tradeoffQuality`), the result and quality will be returned.

```
Input: DataWithQualityVariable data
Input: QualityVector reqQuality
Output: DataWithQualityVariable resultData
Output: QualityVector retQuality
if ∃value st. Quality(value) > reqQuality then
    return value and Quality(value)
else
    if ∃ running Transformer p st. Quality(Result(p)) > reqQuality then
        Wait for p;
        return Result(p) and Quality(Result(p))
    else
        foreach Transformer p applicable to data do
            if QualityResultEstimate(p) > reqQuality then
                if CostEstimate(p) < availResource then
                    foundTransformer = p;
                    break
                end
            end
        end
        if foundTransformer then
            Launch foundTransformer;
            Wait for foundTransformer;
            return Result(foundTransformer) and
            Quality(Result(foundTransformer))
        else
            FindBestMatch;
        end
    end
end
```

Algorithm 2: Basic quality response algorithm for the quality-driven runtime

5.3.3.2 Implementation of extensible semantics

Section 5.3.1 defined a few use cases of the quality approach; this section shows here how the proposed implementation allows these cases to be implemented.

Addition Addition of an additional computation is very easily done with the runtime and is used extensively in the result section for Quake 3. The code for adding a computation is given in Figure 27. The code snippet considers one quality parameter which can take either a value of 0 or 1 depending on whether the additional

```
1 QualityVector qv = (1); /* Corresponds to additional task being
    done */
  globalRuntime->futureQuality(data, &qv, time);
  /* Do some work for time */
  globalRuntime->tradeoffQuality(data, &qv, waitTime);
```

Figure 27: Adding a computation to a program

computation has been performed. The programmer starts by informing the runtime that he will want the additional task run on the data (by specifying that the quality should be 1). Some parallel main task is then performed. The `tradeoffQuality` call asks the runtime to return the result of the computation. If the additional task has completed, the result will be returned immediately. Otherwise, the runtime has the option of waiting for `waitTime`. If after that time, the result is still not available, `data` will be returned unmodified (with a quality of (0)).

Revision Revision is probably the hardest concept for the programmer to implement but can be very powerful. An example based on the MPEG algorithm is given in Figure 28. In the MPEG algorithm, pictures (or frames) can be encoded as I-frames, P-frames or B-frames. The I-frame takes the least time to encode but also produces the least compression. P and B-frames allow temporal compression (by comparing the frame to past and possibly future frames) but require additional work to find the 'motion vector' which identifies how the image has changed. Calculating the motion vector is an expensive process and exhibits a great variation in execution time (the algorithm might find the motion vector right away or it might have to search the entire space). The example shows the encoding of a P-frame with some filtering. In the example one can see that the input given at the time the call to `futureQuality` is made changes. The Transformer that was launched to meet the `futureQuality` request is thus operating on older data which may not produce the highest quality result. The runtime therefore makes the change available to the Transformer which

```
1 QualityVector qv = (1); /* Corresponds to find_motion being done
    */
  globalRuntime->setInput(data, userInput);
  globalRuntime->futureQuality(data, &qv, time);
  calculate_filter(userInput->getInput->block1);
  globalRuntime->updateInput(data, userInput);
6 calculate_filter(userInput->getInput->block2);
  globalRuntime->updateInput(data, userInput);
  globalRuntime->tradeoffQuality(data, &qv, waitTime);
```

Figure 28: Quality-driven MPEG encoding algorithm

is then responsible for checking whether new inputs are available. While this does put the burden on the programmer, it also allows great generality and flexibility. The Transformer can ignore any input change or partially take them into consideration.

Refinement Refinement is a concept completely implemented by the runtime. Consider the example of calculating Taylor expansion terms again. If a programmer-defined thread `A` requires an object `foo` to be of quality 10 (with 10 terms used) and a programmer-defined thread `B` requires the same object to be of quality 20. Originally, both threads have `foo` of quality 0. When thread `A` makes a call to the runtime, a Transformer to calculate the first 10 terms is launched. When thread `B` makes a call to the runtime, the runtime will notice that the first 10 terms are being calculated by another Transformer. It will then look for a Transformer capable of bringing the quality from 10 to 20 and compare it with a Transformer capable of bringing the quality from 0 to 20. In this case, it will most likely determine that it is better to wait for the result from the Transformer already running and pipe it to another Transformer to meet `B`'s request. Figure 29 shows this.

The strength of the runtime thus lies in the fact that it is capable of sharing results from other computations, possibly launched in other threads to accelerate the computation of future tasks. This does require some support from the Transformer

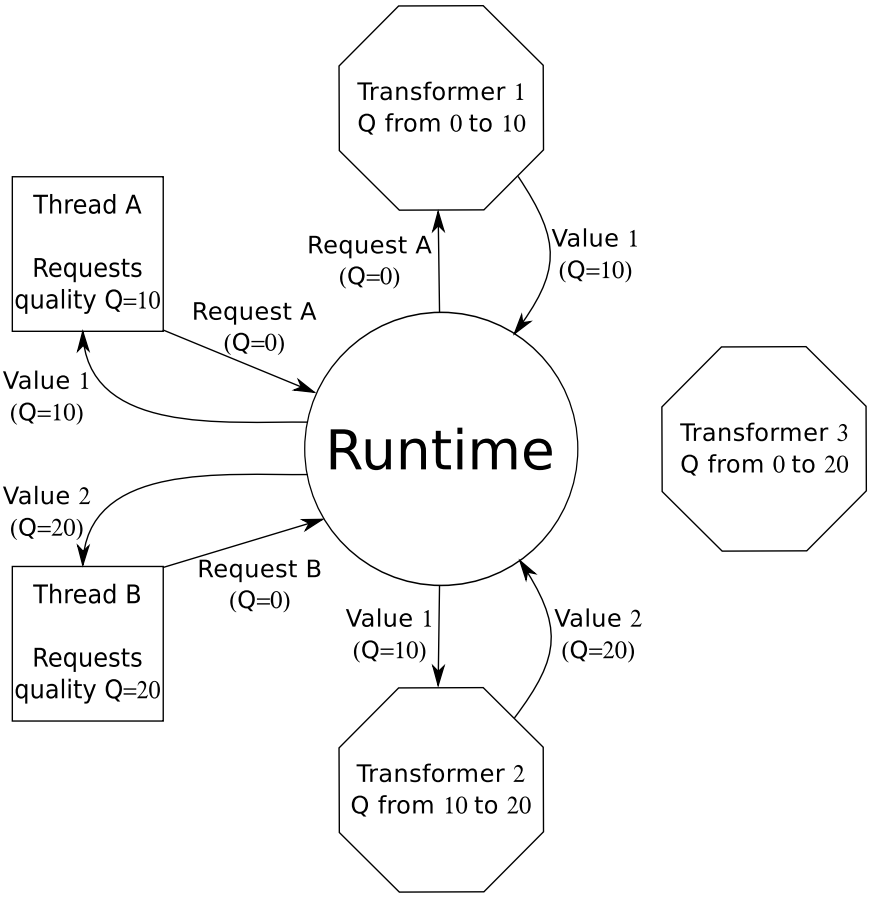


Figure 29: Refinement of a computation. Thread `A` requests the result first. The octagons represent transformers. In this example, the runtime will ask Transformer 1 to produce Value 1 which will then be used to compute the result (Value 2) for thread `B`. Transformer 3 is not used as a result of quality 10 is already available to use.

objects and they have to be written to be extensible. In Figure 29, the three Transformers may actually be one and the same with intelligent quality estimator and cost estimator functions. The runtime will present all the possible values that it has access to (current and in progress) as base input to the estimator functions of all the Transformers. This allows the Transformers to determine the estimated produced quality and cost based on the quality of the value that it will be passed in.

Morphing Morphing is intrinsically supported by the runtime as it chooses a Transformer to improve quality based on quality requirements but also resource constraints. The computations launched by the runtime to meet the quality requirements can thus

be radically different depending on resource availability. Figure 30 illustrates this with the coding of a MPEG frame. Supposing the programmer defines 3 Transformer

```
QualityVector qv = (1); /* Signifies produce at least an I-Frame
    */
2
globalRuntime->tradeoffQuality(frameData, &qv, availTime);
```

Figure 30: Program morphing

objects, one calculating an I-frame, another a B-frame and a third a P-frame, the runtime can dynamically choose which one to run based on the resource availabilities and the time constraint given by the programmer. Here one sees that the main program, which will be blocked until one of the Transformers finishes calculating, will take on one of three possibilities.

5.3.3.3 Runtime strengths

To summarize, the strengths of the runtime are the following:

- Avoid redundant calculations by re-using results produced by similar computations.
- Construct pipes of execution by allowing the result of one computation to be used as the input to another computation.
- Allow computations to iteratively refine their results if new inputs become available.
- Abstract away from the programmer the issues related to data sharing. The runtime takes care of passing arguments by copy semantics and synchronizing the copies at appropriate places (when a quality requirement is made).

5.4 Experimental results

5.4.1 Quake 3 description

To demonstrate the quality approach, the popular First Person Shooter (FPS) game Quake 3 was modified.

Quake 3 is a multi-player FPS game (totaling about 285000 lines of C code) in which the player's character moves in a virtual world called a map, interacting with objects in it (such as picking up weapons, power-ups etc.). The goal of the player is to score points by "killing" a virtual enemy (either computer controlled bots or characters controlled by other human players). The Quake 3 game is built upon the Quake 3 game engine which consists of the core of the game play and other functionality. In the general sense, a game engine consist of many different modules [50, 51, 7]. The most complete game engines, like the one used in Quake 3, include rendering—to draw the scene onto the screen—, a physics engine—to describe the physical interactions—, an AI engine—to control the bots—, sound support and other lower level components to support scripting and networking for example.

The possibilities of extending Quake 3's semantics are numerous in the physics engine (to reduce "physical impossibilities" for example) and AI engine (to add more "thinking" like collaboration, path planning, etc.) in particular.

Quake 3 was selected as a demonstration vehicle as a version of the code is freely available [83] which allowed easy integration with the quality framework. Quake 3 is also a full fledged game engine which made the experiments realistic.

Note that in [86], Zyulkyarov et al. seek to add parallelism to Quake through STMs thereby allowing Quake to run on multi-core platforms. This work is different in the sense that, while it uses the same demonstration vehicle, it exploits variable semantics in game engines to extract parallelism.

5.4.2 Experimental setup

In the experiments, 'quality' represents whether or not certain additional effects were run. In this sense, the experiments mostly used the 'addition' programming construct to opportunistically improve the game. The measured frame rate was used to determine the available time for computations.

Since Quake is written in C, a modified version of the quality framework was implemented to be compatible with it. Newer games are written in C++ and the framework would fit more easily in those. However, the source code for those games is not publicly available.

The modified version of Quake was run on an Intel quad-core Xeon. Note that given the implemented effects, at most one extra core was utilized. For each run, Quake was played for two minutes in full-screen mode with other bots. When the quality framework was used, the runtime aimed to maintain 65 FPS [1].

Four different effects that modified the behavior of rocket and grenade projectiles to improve their quality were implemented. The words 'effect' and 'quality improvement' are interchangeably used through this section.

In the original version of Quake 3, rocket projectiles are not affected by gravity, they always explode on impact, and do not employ any homing logic. Three effects to improve this default behavior were implemented: **i)** rockets are subject to gravity, **ii)** rockets can "bounce" off of walls and **iii)** rockets can dynamically identify and home in on adversaries. The fourth effect has to do with grenades which can now be "attracted" to nearby opponents.

Note that these effects are for illustration purposes only. A game programmer, with his more intimate knowledge of the game, may have chosen other effects.

[1] Note that since Quake is an old game, rendering details were maximized so that the baseline case (without any effects and without the quality framework) ran at 70-75 FPS.

5.4.3 Results

Table 6: Mean and standard deviation of the FPS (frame-per-second) for different quality improving effects. UP means that the baseline version of Quake only had one thread and SMP means that the rendering thread was separate.

(a) One Quake thread (UP)

Effects	**Inline** *UP*		**Quality Driven** *UP*	
	Mean	StdDev	Mean	StdDev
None	72.4	9.62	N/A	N/A
Gravity	70.4	10.47	72.8	7.39
Bounce	59.7	11.25	72.6	9.46
Gravity + Bounce	56.9	12.93	73.5	8.44
Gravity + Bounce + Homing	56.5	11.52	66.6	9.57
Gravity + Bounce + Homing + Vortex	56.1	12.58	68.3	9.39

(b) Two Quake threads (SMP)

Effects	**Inline** *SMP*		**Quality Driven** *SMP*	
	Mean	StdDev	Mean	StdDev
None	70.6	12.04	N/A	N/A
Gravity	69.3	9.98	68.2	9.6
Bounce	64.6	12.50	67.3	9.68
Gravity + Bounce	54.6	12.21	68.7	7.82
Gravity + Bounce + Homing	54.2	11.87	70.7	7.63
Gravity + Bounce + Homing + Vortex	53.7	11.7	70.3	8.48

The quality framework's goal is to improve the quality of the computations without compromising performance. The results will also show that the framework allows for better resource usage by occupying more of the available resources when possible. Videos showing the effects are available at `http://www.cc.gatech.edu/~romain/IPDPS_Q3`.

5.4.3.1 Performance is maintained

Results concerning performance are presented in Table 6. The tests were run with the various effects turned on for both the uni-processor version (where there is no separate rendering thread) and the SMP version of Quake (where rendering is done

in a separate thread). The "inline" columns show results when the effects were always 'on': quality is therefore maximized at the expense of performance. The 'qual. driven' columns show results using the quality framework when the effects are dynamically activated depending on resource and time availability.

Several conclusions can be drawn from these results. First, inlining all special effects severely impacts user experience as the frame rate drops significantly (by around 23% for both the UP and SMP versions). Opportunistically running the special effects with the quality framework does not cause such a drastic degradation in frame rate. With all effects opportunistically activated, the degradation in FPS was less than 1% in the SMP case, and about 5% in the UP case.

5.4.3.2 Quality is improved

In the quality-driven approach, the frame rate is maintained in part because the runtime decides if and when to launch a quality improving computation based on the amount of resource and time available. Table 7 shows the proportion of effects asked for by Quake that were actually launched for the SMP version of Quake with all effects enabled.

Note that in Table 7 the total number of canceled effects is much higher than the sum of individual canceled effects due to an implementation artifact where the type of canceled effects is not always known. In other words, the total number of canceled effect is accurate but the exact distribution is unknown. The results show that around

Table 7: Breakdown of effects that were 'unlaunched', 'executed' and 'canceled' for the SMP version of Quake. No effects were culled after they started executing.

Effects	Total	Unlaunched	Executed	Canceled
Gravity	228	160	68	0
Bounce	744	420	160	164
Homing	2003	1233	766	4
Vortex	2273	1037	1235	1
All	6572	2850	2229	1493

34% of all effects were executed. 43% were not launched due to a lack of time and 23% were canceled due to the game play no longer needing them (misprediction).

It is also interesting to note that some of the more expensive effects (such as bouncing[2]) are executed much more infrequently (only 21% of the times) than effects that require fewer resources such as gravity (29% of the times) or even the grenade effect (54% of the times). Since equal weights were given to each of the effects, this is consistent with the runtime's goal to maximally improve performance without degrading performance. Effects that are 'cheaper' and give the same quality improvements as more 'expensive' ones will be favored.

5.4.3.3 FPS evolution

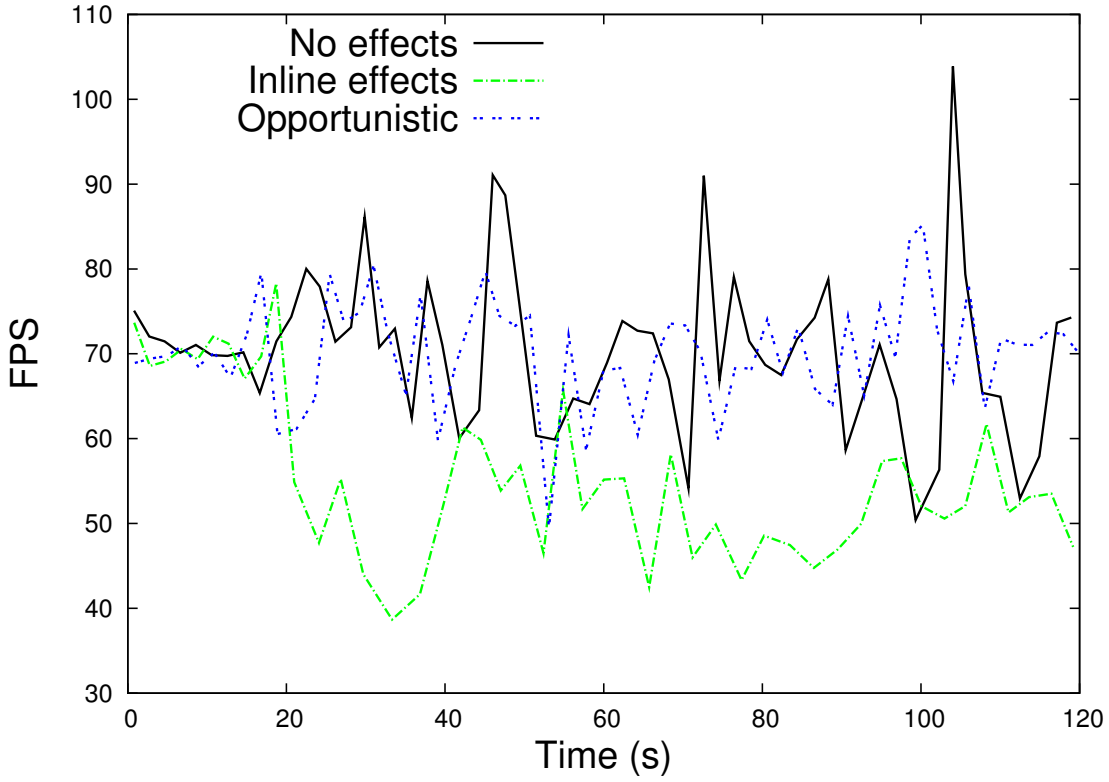


Figure 31: Evolution of the frame rate in Quake

[2]Rocket bouncing is expensive as it increases the 'life' of the missile and the game therefore has to track more objects.

Figure 31 shows the evolution of the frame rate for the execution of the SMP version of quake with **i)** no effects, **ii)** all effects inlined and **iii)** effects opportunistically added using the quality framework. This corresponds to the data summarized in the first and last rows of Table 6.

The results show that in the opportunistic case, the frame rate is slightly lower but smoother than with no effects at all. This shows that the runtime is taking advantage of the opportunities presented by a higher frame rate by running quality improving computations. Note that when all effects are inlined, the frame rate is consistently lower than the other two cases.

5.4.3.4 Framework overhead

The memory impact of the modifications to Quake was measured, specifically the non-swapped physical memory usage (size of text-resident and data-resident sets). The baseline Quake program (UP version) used 81 MB of RAM. Inlining all effects required 82 MB of RAM. With the quality approach, running those effects caused an increase of 6.2% to 87 MB. This increase is mostly due to the bookkeeping overhead (in the runtime), the copy semantics and the thread states that have to be kept for context switching.

Finally, the time the main thread waits for results from the Transformers was also measured. Out of a total runtime of 2 minutes, the main thread waits for a result for less than 93 milliseconds which shows that the runtime does not significantly slow down the main thread and correctly determines when to opportunistically launch an effect and when not to.

5.5 Related work

This section will first describe the state of the art in dynamic and adaptive Quality of Service systems. It will then summarize related work in parallel programming models as applicable to our system. It will also briefly describe related work in real-time

systems.

5.5.1 Adaptive QoS

There is a significant body of literature on application, middle-ware and operating system level frameworks for supporting dynamic Quality of Service [25, 18, 40], especially in soft real-time environments. The Rialto Operating System [42, 41] provides programming abstractions that allow multiple real-time and non real-time programs to execute concurrently and share resources. In [59], the authors propose a system based on a Fair Share scheduling algorithm in which programs are allotted resources (specifically the CPU) depending on the availability of those resources and the "importance" of that program relative to all other programs currently running. In yet another work [10], the authors propose an execution model where a program has multiple execution levels and can switch between them depending on the resources that have been allocated to it by the soft real-time scheduler.

This work differs from the ones above in several aspects. Firstly, the presented system is based on "variable semantics" - the program can, depending on resource availability, adjust its own functionality. The "execution repertoire" described in [42] is the closest to this notion, although not as dynamic. Work by Kumar et al. also exploits variable semantics [47] but presents an controller-based system to dynamically tweak the application to meet soft real-time deadlines. Kumar does not focus on exploiting parallel resources.

Secondly, the emphasis in these works is on orchestrating all concurrently executing programs such that their real-time constraints (if they have any) are satisfied. In the presented system however, the emphasis is on dynamically exploiting resource availability.

Finally, all the above works on dynamic QoS have been proposed in the context of multiple, largely independent programs executing concurrently in a system. In

contrast, the presented system has been developed in the context of a single program with multiple cooperating threads, where all of the threads share a significant amount of data. This distinction results in the objectives and mechanisms of the runtime system being drastically different than in the other works, especially in the way a result used and produced by a Transformer (a thread) affects the future launching of other Transformers.

5.5.2 Parallel Programming Models and Languages

Extensive work has been carried out (and is also underway) in the parallel computing community on programming models that support the parallelization of applications. Such programming models include sequential languages augmented with threading libraries (C with Pthreads [49] or MPI [34]), programming language extensions that add parallel semantics to sequential languages (C with OpenMP [21]), and programming languages with inherently parallel semantics (X10 [14]). While parallel programming models have had significant success in the scientific computing community, a lot of the desktop/home computing domains like multimedia and gaming are extremely compute-intensive but are not able to make use of the multiple cores. The parallelism inherent in these applications is often either too unstructured to express using OpenMP or too complex to express using a threading library like Pthreads or through message passing between threads/processes as in MPI which force the programmer to handle all the low level synchronization issues between threads.

The proposed approach relies instead on the imprecise nature of the computation performed by such applications in order to utilize the additional processor cores. The imprecision can be in terms of **i)** the numeric accuracy of results, **ii)** the sophistication of models employed, and **iii)** whether parts of the computation can be skipped from time-to-time. Imprecision is captured in the notion of quality in the model.

5.5.3 Soft Real-time Systems

The real-time systems work [9, 59, 28] differs from the proposed approach because the emphasis there is on orchestrating the real-time tasks in a manner that gets all the tasks executed by their deadlines, and not on exploiting parallelism or the dynamic availability of compute resources. This often leads to poor resource utilization in real-time applications. The real-time model is suited for mission and safety critical applications whereas our approach is well-suited for scaling the user experience as more compute resources (multicores) become available. Some work has been done in the real-time community on specifying optional parts of a task and on scheduling the mandatory and optional parts to meet their deadlines [23]. However, this work only deals with tasks as abstract entities and mainly focuses on their scheduling aspects and does not deal with programming abstractions, such as the notion of quality. In particular, the proposed framework allows the definition of a larger class of imprecise semantics using concepts such as refinement to iteratively improve the accuracy of results, extension to extend the computation to a larger size of data and revision to update results using more up-to-date versions of data.

5.6 Conclusion

This chapter demonstrated a new approach to enhancing 3D video games through the exploitation of parallel resources. This new approach allows the specification of scalable semantics in applications that can be enriched and thus adapt to the amount of available resources at runtime. This chapter proposes a C++ API allowing the programmer to define how quality is associated with data. It provides mechanisms for the programmer to dynamically modify his program through quality requirements. Interesting domain specific optimization possibilities are offered by the API and its implementation. This infrastructure is shown to be useful to enrich a well known game called Quake 3 on an Intel quad-core Xeon. The results show that it is possible

to add many effects extending the semantics when a second core is available without degrading the frame rate which measures the user experience.

5.6.1 Thesis discussion

The results presented in this chapter show that, for applications that have variable semantics, it is possible to utilize idling resources to speculatively launch quality-enhancing additional ways thereby improving the overall computation if there is sufficient time. The quality driven framework presented in this chapter exploits *variable semantics* which is an algorithmic property present in many emerging application domains (games and multimedia). The framework allows a variable semantic application to utilize all available parallel resources to provide the best possible quality to the end-user. This therefore proves the thesis statement since the property of variable semantics is used to occupy idling cores to improve the quality of the result.

CHAPTER VI

RELATED WORK

The related works specific to the N-way model, the data-structure driven framework as well as the quality driven framework were discussed in each of those individual chapters. This chapter covers common related work. In particular, this chapter covers the related work in the area of parallel programming models and the solutions that have been explored to address the problem of the sequential portions of applications and the parallelization of irregular algorithms.

6.1 Addressing the sequential bottleneck

Computer architects have, over the years, greatly improved the sequential performance of chips by (among other things):

- Increasing the operating frequency;
- Removing "false" dependencies ('write after write' and 'write after read') through the use of a reorder buffer;
- Improving branch prediction which allows a processor to speculate past a branch and therefore increases the number of instructions executed per second. Architects have also used predictive techniques to predict memory dependencies between memory operations further improving performance;
- Improving code and data pretetching thereby allowing processors to not be held back by the relatively slower memory subsystems.

However, these improvements have been restricted to making a single core faster; the issue of utilizing multiple cores to improve the performance of sequential codes has

not been addressed. Furthermore, very few *programming models* address this issue.

6.1.1 Programming models to improve sequential execution

Work by Trachsel et al. [77, 78] did address this in a way similar to the presented N-way work. However, the model they introduce, called CPE (Competing Parallel Execution) is akin to the "brute force" N-way model and does not address the issue of resource waste. The N-way model introduces a mathematically sound learning model to best estimate the benefits that can be derived from concurrently launching multiple instances of the same computation.

6.2 Expressing parallelism in irregular algorithms

Much more work has gone into making the expression of parallelism in irregular algorithms simpler and more intuitive for the programmer.

6.2.1 The Galois programming model

The Galois programming model [46, 55] also exploits algorithmic properties to improve the parallelization of irregular algorithms. The Galois model was the first to directly tackle the problems associated with irregular algorithms.

The algorithmic property exploited by the Galois model is *commutativity*: the programmer defines whether or not two operations commute semantically. If two operations commute, they can happen in parallel, and if not, they must be serialized. The Galois model therefore raises the level of abstraction as far as conflict detection is concerned: instead of relying on conflict detection at the memory level (either at a byte or object level), the Galois model detects conflict at the operations level. The differences between the Galois model and the data-structure driven framework developed in this thesis were described in Section 3.6.

6.2.2 Concurrent Collections

The Concurrent Collections model (or CnC) [44] , developed by Intel, allows programmers to adopt a more stream-oriented view of a computation. In CnC, programmers define 'steps' that need to be executed as well as data elements that are consumed and produced by each of these steps. The execution of a step generates more steps and data elements which will, in turn, lead to more step executions.

The CnC model therefore allows programmers to think solely about dependencies between steps in a way that is much more akin to the natural diagram flow of a program. A runtime system manages the actual discovered parallelism freeing the programmer from having to explicitly specify the presence or absence of parallelism.

6.2.3 Analysis based approaches

Other approaches such as DPJ and Jade (both described in Section 3.6) seek to determine the amount of parallelism either statically or dynamically through program analysis. These approaches are similar to the ones developed in this thesis but do not seek to exploit algorithmic properties relying instead of program analysis and shifting the weight away from the programmer to a more fixed approach. While it is frequently advantageous to free the programmer from certain tasks, this thesis motivates that allowing the expression of knowledge that the programmer already has is beneficial and can be usefully exploited.

CHAPTER VII

CONCLUSION

As increasing parallel resources become more and more common in the end-user arena, programmers will have to adapt to this new programming landscape and devise techniques to continuously offer more or better features to the user. Programmers can no longer enjoy the free ride given to them by computer architects: frequency scaling is no longer possible and applications can no longer piggy-back on increases in frequency.

To take advantage of the increase in parallel resources, programmers have to solve the dual problem of usefully utilizing idling resources and addressing the bottleneck of hard to parallelize or sequential codes. This thesis demonstrates that traditional parallel programming techniques that rely on the "breaking-up" of computations (such as data and task parallelism) are not sufficient to address these problems for a large class of applications. Instead, this thesis shows how algorithmic properties can be exploited to devise novel programming models that solve both problems. In particular, the N-Way programming model exploits *algorithmic diversity* to speculatively launch competing ways to perform the same computation picking the best one just in time. The quality driven computing model exploits *variable semantics* to speculatively launch additional quality improving ways that are, time permitting, merged back into the main computation thereby providing maximal quality within the constraints of the platform. Finally, *data-structure semantics* can be expressed and exploited to improve the performance of optimistic parallelism. This last model applies specifically to computations that are hard to parallelize (a 'may' dependency).

All three models presented enable applications to *opportunistically* scale with the

increasing platform resources; the application will adapt to the amount of parallelism that it can exploit and perform better on more parallel platforms irrespective of whether it is embarrassingly parallel or hard to parallelize. Going forward, similarly novel techniques will have to be developed for other classes of applications; this thesis deals only with computations that exhibit one of the algorithmic properties discussed (diversity, variable semantics and data-structure semantics).

7.1 Future work

This thesis opens up multiple areas of interesting research. This section describes some of the possible future work. Refer to the specific 'Future work' sections of each chapter for more detail. This section summarizes the main points and provides overarching future work.

7.1.1 N-way framework

Different paths of research are opened by the N-way framework.

7.1.1.1 Parallelism versus N-way

The N-way model is currently designed to exploit parallel resources only through the expression of diversity. However, it would be interesting to study the implications of having both traditional parallelism (task/data) and N-way parallelism in the same application. Several challenges exist in trying to determine the optimum "mix" of traditional parallelism versus N-way parallelism. Furthermore, one could also consider the implications of having multiple parallel implementations of the same computation: each parallel implementation could constitute a way. Indeed, given the complexity of parallel programming and the unpredictable interleavings which may occur, there is a large diversity due to the parallel implementations. This added diversity could be exploited by N-way with the caveat that each way now uses many more resources and the model is therefore potentially more wasteful. More aggressive culling and more

precise and accurate learning schemes would be required to compensate for this.

7.1.1.2 Power implications and thread collaboration

The power implications of N-way could also be explored. Intel recently implemented in its Atom chips, the principle of 'race to idle' where the chip tries to execute as quickly as possible (using the most energy) so that it finishes its work as soon as possible thereby enabling it to sleep. The N-way model is ideal for the 'race to idle' as it uses all available resources for a shorter period of time. We have already done some initial exploration of the power implications of the N-way model in [17] but this could be pushed much further. In particular, the N-way model could be given a power envelope to respect and have to work within those constraints. One could also imagine that the results of the ways that are thrown away could be somehow reintegrated in the committed results thereby reducing the waste of the work that was done. We have collaborated on some initial work on thread collaboration [65] which could be used to utilize the results of the thrown-away ways.

7.1.1.3 Improving the learning model with static information

Currently, the N-way model only relies on the information it learns at runtime about the execution time distribution to determine the set of ways to launch. This could be complemented by static analysis trying to predict the execution time of a particular way. Some relevant work includes [1, 36]. Such a prediction could be used to guide the learning algorithm in making better choices by providing it with more information.

7.1.2 Profiling in the symbolic dataspace

This thesis utilized the information gained from the patterns extracted in the symbolic dataspace to determine the potential overlaps between the data footprints of operations. This work could be extended to better classify patterns, for example by adding context sensitivity thereby allowing for the access patterns to be more precise.

However, understanding access patterns in the symbolic dataspace has implications that go beyond the application presented in this thesis.

7.1.2.1 Application signature

The fact that the access patterns are based on programmer attributed names means that if the programmer assigns the same names to different sets of variables in different applications where the variables of the same name play a similar role in the applications (for example, the "input" or the "output") the access patterns of these applications can be compared and a signature for the applications can be created. This signature could be used to:

- Determine the amount of diversity among the applications: applications with the same access pattern would be similar and therefore may not exhibit great diversity.
- Classify applications: if an application with a certain signature behaves in a particular way on a given system, it could be expected that other applications with similar signatures would also behave in a similar way. This could be useful in determining whether an application will run well on a particular system.

7.1.2.2 Bug detection

A common software engineering technique to find bugs consists in taking known good runs of an application and known bad runs of the same application and finding differences between traces of the runs. One could compare the memory maps of the good runs with that of the bad runs at each point in the execution. This would show which variables or which aliasing pointers are different and therefore potential candidates for bug causes.

7.1.2.3 Re-locating memory

In large systems, the cost of data movement in terms of energy is non negligible. It is therefore important to try to minimize such movement. Being able to extract patterns could enable a runtime to move memory around in such a way that the memory that will be touched is close by. This could be used in GPUs for example where memory accesses that are not well structures are a huge problem. Indeed, GPUs work very well for blocked memory accesses but if the memory accesses diverge (due to pointer based structure for example), the memory bandwidth of GPUs goes down dramatically. With the described tool, the memory space could be reorganized such that block accesses are again possible.

7.1.3 Final thoughts

For many of the future works described above, collaboration with other areas would be key. In particular, for the dataspace profiling future work, collaboration with OS researchers and architecture researchers would be crucial in determining how the hardware and OS could both contribute to better understanding the access patterns (through information from the page tables for example or from the memory subsystem). We believe that the dataspace profiling work has a great deal of potential applications as it allows the programmer to understand his program in a very different way, bringing back structure to what was a very unstructured memory access patterns.

We hope that this thesis will serve as a starting point to future research into gaining a better understanding of applications through their algorithmic properties.

REFERENCES

[1] Aleen, F., Sharif, M., and Pande, S., "Input-driven dynamic execution prediction of streaming applications," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, (New York, NY, USA), pp. 315–324, ACM, 2010.

[2] Allen, M. D., Sridharan, S., and Sohi, G. S., "Serialization sets: a dynamic dependence-based parallel execution model," in *PPoPP '09*, (New York, NY, USA), pp. 85–96, ACM, 2009.

[3] Ansel, J., Wong, Y. L., Chan, C., Olszewski, M., Edelman, A., and Amarasinghe, S., "Language and compiler support for auto-tuning variable-accuracy algorithms," in *CGO '11*, IEEE Computer Society, 2011.

[4] ARM, "Cortex-a9 processor." `http://www.arm.com/products/processors/cortex-a/cortex-a9.php`.

[5] Asanovic, K. and others, "The landscape of parallel computing research: A view from berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[6] Berger, E. D., Yang, T., Liu, T., and Novark, G., "Grace: safe multithreaded programming for C/C++," in *OOPSLA '09*, (New York, NY, USA), pp. 81–96, ACM, 2009.

[7] Bishop, L., Eberly, D., Whitted, T., Finch, M., and Shantz, M., "Designing a pc game engine," *IEEE Computer Graphics and Applications*, vol. 18, pp. 46–53, January 1998.

[8] Bocchino, Jr., R. L., Adve, V. S., Dig, D., Adve, S. V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., and Vakilian, M., "A type and effect system for deterministic parallel java," in *OOPSLA '09*, (New York, NY, USA), pp. 97–116, ACM, 2009.

[9] Bollella, G. and Jeffay, K., "Support for real-time computing within general purpose operating systems-supporting co-resident operating systems," in *Proceedings of the Real-Time Technology and Applications Symposium*, RTAS '95, (Washington, DC, USA), pp. 4–, IEEE Computer Society, 1995.

[10] Brandt, S., Nutt, G., Berk, T., and Humphrey, M., "Soft real-time application execution with dynamic quality of service assurance," in *Sixth International Workshop on Quality of Service (IWQoS 98)*, pp. 154–163, 1998.

[11] Budimlic, Z., Burke, M., Cavé, V., Knobe, K., Lowney, G., Newton, R., Palsberg, J., Peixotto, D. M., Sarkar, V., Schlimbach, F., and Tasirlar, S., "Concurrent collections," *Scientific Programming*, vol. 18, no. 3-4, pp. 203–217, 2010.

[12] Cachopo, J. and Rito-Silva, A., "Versioned boxes as the basis for memory transactions," *Sci. Comput. Program.*, vol. 63, no. 2, pp. 172–185, 2006.

[13] Cao Minh, C., Chung, J., Kozyrakis, C., and Olukotun, K., "STAMP: Stanford transactional applications for multi-processing," in *IISWC '08*, September 2008.

[14] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V., "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA '05*, (New York, NY, USA), pp. 519–538, ACM Press, 2005.

[15] "CLANG: A C family frontend for LLVM." `http://clang.llvm.org/`, 2010.

[16] Cledat, R., Kumar, T., Sreeram, J., and Pande, S., "Opportunistic computing: A new paradigm for scalable realism on many cores," in *HotPar 2009: 1st USENIX Workshop on Hot Topics in Parallelism*, USENIX, 2009.

[17] Cledat, R. and Pande, S., "Energy efficiency via the n-way model," in *PESPMA 2010, in conjunction with ISCA*, ACM, 2010.

[18] Compton, C. L. and Tennenhouse, D. L., "Collaborative load shedding for media-based applications," in *International Conference on Multimedia Computing and Systems*, pp. 496–501, 1994.

[19] Costa, S., "Game engineering for a multiprocessor architecture," Master's thesis, John Moores University, Liverpool, 2004.

[20] Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-tuong, A., and Hiser, J., "N-variant systems: A secretless framework for security through diversity," in *In Proceedings of the 15th USENIX Security Symposium*, pp. 105–120, 2006.

[21] Dagum, L. and Menon, R., "Openmp: an industry standard api for shared-memory programming," *Computational Science and Engineering, IEEE*, vol. 5, no. 1, pp. 46 – 55, 1998.

[22] de Gelas, J., "The quest for more processing power: Multi-core and multi-threaded gaming." `http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2377&p=3`, March 2005.

[23] de Oliveira, R. S., da Silva Fraga, J., and Farines, J.-M., "Scheduling imprecise tasks in real-time distributed systems," in *ISORC '01*, pp. 319–326, IEEE Computer Society, 2001.

[24] "Dimacs benchmarks." `http://tinyurl.com/myj2m7`, 2009.

[25] Fan, C., "Realizing a soft real-time framework for supporting distributed multimedia applications," in *Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, FTDCS '95, (Washington, DC, USA), pp. 128–, IEEE Computer Society, 1995.

[26] Felber, P., Fetzer, C., and Riegel, T., "Dynamic performance tuning of word-based software transactional memory," in *PPoPP '08*, (New York, NY, USA), pp. 237–246, ACM, 2008.

[27] Gall, D. L., "Mpeg: a video compression standard for multimedia applications," *Commun. ACM*, vol. 34, no. 4, pp. 46–58, 1991.

[28] Gopalakrishnan, R. and Parulkar, G. M., "Bringing real-time scheduling theory and practice closer for multimedia computing," in *Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '96, (New York, NY, USA), pp. 1–12, ACM, 1996.

[29] Greenlaw, R., Hoover, H. J., and Ruzzo, W. L., "A compendium of problems complete for p," 1991.

[30] Greenlaw, R., Hoover, H. J., and Ruzzo, W. L., *Limits to parallel computation : P-completeness theory*. New York, NY, USA: Oxford University Press, 1995.

[31] Hamadi, Y., Jabbour, S., and Sais, L., "Manysat: Solver description," Tech. Rep. MSR-TR-2008-83, Microsoft Research, May 2008.

[32] Harris, T. and Fraser, K., "Language support for lightweight transactions," in *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, (New York, NY, USA), pp. 388–402, ACM Press, 2003.

[33] Harris, T. and Fraser, K., "Language support for lightweight transactions," in *OOPSLA '03*, (New York, NY, USA), pp. 388–402, ACM, 2003.

[34] Hempel, R., "The mpi standard for message passing," in *HPCN Europe 1994: Proceedings of the nternational Conference and Exhibition on High-Performance Computing and Networking Volume II*, (London, UK), pp. 247–252, Springer-Verlag, 1994.

[35] Hill, M. D. and Marty, M. R., "Amdahl's law in the multicore era," *IEEE COMPUTER*, 2008.

[36] Huang, L., Jia, J., Yu, B., Chun, B.-G., Maniatis, P., and Naik, M., "Predicting execution time of computer programs using sparse polynomial regression," in *Advances in Neural Information Processing Systems 23* (Lafferty,

J., Williams, C. K. I., Shawe-Taylor, J., Zemel, R., and Culotta, A., eds.), pp. 883–891, 2010.

[37] "Intel haswell." http://tinyurl.com/28dxp67, 2010.

[38] "Intel shows 48-core 'datacentre on a chip'." http://tinyurl.com/2fyhejo, 2010.

[39] Iyer, S. K., Jain, J., Prasad, M. R., Sahoo, D., and Sidle, T., "Error detection using BMC in a parallel environment," in *CHARME*, pp. 354–358, 2005.

[40] Jensen, E., Locke, C., and Tokuda, H., "A time driven scheduling model for real-time operating systems," 1985.

[41] Jones, M. B., Leach, P. J., Draves, R. P., and Barrera, III, J. S., "Modular real-time resource management in the rialto operating system," in *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, (Washington, DC, USA), pp. 12–, IEEE Computer Society, 1995.

[42] Jones, M. B., McCulley, D. L., Forin, A., Leach, P. J., Roşu, D., and Roberts, D. L., "An overview of the rialto real-time architecture," in *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, EW 7, (New York, NY, USA), pp. 249–256, ACM, 1996.

[43] Kecher, R., "Futures: Asynchronous invocation." http://cplusplus.co.il/2010/05/31/futures-asynchronous-invocation/.

[44] Knobe, K., "Ease of use with concurrent collections (CnC)," in *HotPar 2009: 1st USENIX Workshop on Hot Topics in Parallelism*, USENIX, 2009.

[45] Kuffner Jr., J. J. and Lavalle, S. M., "RRT-connect: An efficient approach to single-query path planning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, pp. 995–1001, 2000.

[46] Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., and Chew, L. P., "Optimistic parallelism requires abstractions," in *PLDI '07*, pp. 211–222, 2007.

[47] Kumar, T., Cledat, R. E., and Pande, S., "Dynamic tuning of feature set in highly variant interactive applications," in *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, (New York, NY, USA), pp. 289–298, ACM, 2010.

[48] Larus, J. R. and Rajwar, R., *Transactional Memory*. Morgan and Claypool, 2006.

[49] Lewis, B. and Berg, D. J., *Multithreaded programming with Pthreads*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.

[50] Lewis, M., "The new cards," *Commun. ACM*, vol. 45, no. 1, pp. 30–31, 2002.

[51] Lewis, M. and Jacobson, J., "Introduction," *Commun. ACM*, vol. 45, no. 1, pp. 27–31, 2002.

[52] "LLVM: A low-level virtual machine." `http://llvm.org`, 2011.

[53] Luby, M. and Ertel, W., "Optimal parallelization of las vegas algorithms," in *STACS '94*, pp. 463–474, Springer, 1994.

[54] Mandviwala, H. A., Ramachandran, U., and Knobe, K., "Languages and compilers for parallel computing," ch. Capsules: Expressing Composable Computations in a Parallel Programming Model, pp. 276–291, Berlin, Heidelberg: Springer-Verlag, 2008.

[55] Mendez-Lojo, M., Nguyen, D., Prountzos, D., Sui, X., Hassan, M. A., Kulkarni, M., Burtscher, M., and Pingali, K., "Structure-driven optimization for amorphous data-parallel programs," in *PPoPP '10*, (New York, NY, USA), ACM, 2010.

[56] Mitzenmacher, M. and Upfal, E., *Probability and Computing*. Cambridge University Press, 2005.

[57] Motwani, R. and Raghavan, P., *Randomized Algorithms*. Cambridge University Press, 1995.

[58] Mukherjee, S. S., Sharma, S. D., Hill, M. D., Larus, J. R., Rogers, A., and Saltz, J., "Efficient support for irregular applications on distributed-memory machines," in *PPOPP '95*, (New York, NY, USA), pp. 68–79, ACM, 1995.

[59] Nieh, J. and Lam, M. S., "A smart scheduler for multimedia applications," *ACM Trans. Comput. Syst.*, vol. 21, pp. 117–163, May 2003.

[60] Nikhil, R. S., Ramachandran, U., Rehg, J. M., Halstead, Jr., R. H., Joerg, C. F., and Kontothanassis, L. I., "Stampede: A programming system for emerging scalable interactive multimedia applications," in *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '98, (London, UK), pp. 83–99, Springer-Verlag, 1999.

[61] nones, C. G. Q., Madriles, C., Sánchez, J., Marcuello, P., González, A., and Tullsen, D. M., "Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, (New York, NY, USA), pp. 269–279, ACM Press, 2005.

[62] Patterson, D., "The trouble with multicore." http://spectrum.ieee.org/computing/software/the-trouble-with-multicore/, July 2010.

[63] Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M. A., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., and Sui, X., "The tao of parallelism in algorithms," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, (New York, NY, USA), pp. 12–25, ACM, 2011.

[64] Ramachandran, U., Nikhil, R. S., Rehg, J. M., Angelov, Y., Paul, A., Adhikari, S., Mackenzie, K. M., Harel, N., and Knobe, K., "Stampede: A cluster programming middleware for interactive stream-oriented applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, pp. 1140–1154, 2003.

[65] Ravichandran, K., Cledat, R., and Pande, S., "Collaborative threads: Exposing and leveraging dynamic thread state for efficient computation," in *HotPar 2010: 2nd USENIX Workshop on Hot Topics in Parallelism*, USENIX, 2010.

[66] Reid, W., Kelly, W., and Craik, A., "Reasoning about inherent parallelism in modern object-oriented languages," in *ACSC '08*, (Darlinghurst, Australia, Australia), pp. 27–36, Australian Computer Society, Inc., 2008.

[67] Reinders, J., *Intel threading building blocks*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[68] Reinelt, G., "TSPLIB - a traveling salesman problem library," in *ORSA Journal on Computing*, vol. 3, pp. 376–384, 1991.

[69] Rinard, M. C. and Lam, M. S., "The design, implementation, and evaluation of jade," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 3, pp. 483–545, 1998.

[70] Salamat, B., Jackson, T., Gal, A., and Franz, M., "Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space," in *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, (New York, NY, USA), pp. 33–46, ACM, 2009.

[71] Selman, B., Kautz, H., and Cohen, B., "Local search strategies for satisfiability testing," in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pp. 521–532, 1995.

[72] Song, Y., Kalogeropulos, S., and Tirumalai, P., "Design and implementation of a compiler framework for helper threading on multi-core processors," in *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, (Washington, DC, USA), pp. 99–109, IEEE Computer Society, 2005.

[73] Sutter, H., "The free lunch is over." http://www.gotw.ca/publications/concurrency-ddj.htm, 2005.

[74] Sweeney, T., "The next mainstream programming language: a game developer's perspective," in *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (New York, NY, USA), pp. 269–269, ACM Press, 2006.

[75] TomLab, "CPLEX parameters interface." http://tomopt.com/docs/cplexug/tomlab_cplex014.php, March 2010.

[76] Trachsel, O., *Application-level Multi-variant Speculation with Competitive Parallel Execution*. PhD thesis, ETH Zurich, November 2010.

[77] Trachsel, O. and Gross, T., "A platform for competitive execution," in *PESPMA 2008, in conjunction with ISCA*, ACM, 2008.

[78] Trachsel, O. and Gross, T. R., "Variant-based competitive parallel execution of sequential programs," in *CF '10: Proceedings of the 7th ACM international conference on Computing frontiers*, (New York, NY, USA), pp. 197–206, ACM, 2010.

[79] Tygert, M., "A fast algorithm for computing minimal-norm solutions to underdetermined systems of linear equations," May 2009.

[80] Usui, T., Behrends, R., Evans, J., and Smaragdakis, Y., "Adaptive locks: Combining transactions and locks for efficient concurrency," in *PACT '09*, pp. 3 –14, sept. 2009.

[81] Vazirani, V., *Approximation Algorithms*. Springer, 2001.

[82] Wall, M., "GAlib." http://lancet.mit.edu/ga/, 2009.

[83] "Ioquake." http://www.icculus.org/quake3/, 2006.

[84] Wintersteiger, C. M., Hamadi, Y., and Moura, L., "A concurrent portfolio approach to smt solving," in *CAV '09*, (Berlin, Heidelberg), pp. 715–720, Springer-Verlag, 2009.

[85] Zhuang, X., Eichenberger, A. E., Luo, Y., O'Brien, K., and O'Brien, K., "Exploiting parallelism with dependence-aware scheduling," in *PACT '09*, (Washington, DC, USA), pp. 193–202, IEEE Computer Society, 2009.

[86] Zyulkyarov, F., Gajinov, V., Unsal, O. S., Cristal, A., Ayguadé, E., Harris, T., and Valero, M., "Atomic quake: using transactional memory in an interactive multiplayer game server," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, (New York, NY, USA), pp. 25–34, ACM, 2009.

VITA

Romain E. Cledat was born in Pau, France in 1982 to Isabelle and Bertrand, the first of three boys.

Romain spent his childhood both in France and in the United States: he moved to the United States at age 8 and remained there for 5 years living first in Saint Louis and then in New Orleans.

Upon his return to France, Romain finished high-school in a bilingual school in Lyon which allowed him to retain his English fluency and his urge to return to the United States.

Upon graduating from the 'S' series for the Baccalaureate, Romain joined the Lycée du Parc where he completed two years of Classes Prèparatoires before being admitted to the Ecole Centrale de Lyon for an engineering diploma. Upon completing two years, Romain was admitted to Georgia Tech where he earned his Masters in ECE in 2005 and his PhD in Computer Science in 2011.

Romain lives with his wife in Portland Oregon where he works as a Research Scientist for Intel Labs.