© 2009 Mayank Agarwal

IDENTIFYING, QUANTIFYING, EXTRACTING AND ENHANCING IMPLICIT PARALLELISM

BY

MAYANK AGARWAL

B.Tech., Indian Institute of Technology Delhi, 2004

M.S., University of Illinois at Urbana-Champaign, 2006

DISSERTATION

Submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy in Department of Computer Science

in the Graduate College of the

University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Doctoral Committee:

Assistant Professor Matthew Frank, Chair

Professor Sarita Adve

Dr. Pradeep Dubey

Professor Josep Torrellas

Assistant Professor Craig Zilles

ABSTRACT

The shift of the microprocessor industry towards multicore architectures has placed a huge burden on the programmers by requiring explicit parallelization for performance. Implicit Parallelization is an alternative that could ease the burden on programmers by parallelizing applications “under the covers” while maintaining sequential semantics externally. This thesis develops a novel approach for thinking about parallelism, by casting the problem of parallelization in terms of instruction criticality. Using this approach, parallelism in a program region is readily identified when certain conditions about fetch-criticality are satisfied by the region. The thesis formalizes this approach by developing a criticality-driven model of task-based parallelization. The model can accurately predict the parallelism that would be exposed by potential task choices by capturing a wide set of sources of parallelism as well as costs to parallelization.

The criticality-driven model enables the development of two key components for Implicit Parallelization: a task selection policy, and a bottleneck analysis tool. The task selection policy can partition a single-threaded program into tasks that will profitably execute concurrently on a multicore architecture in spite of the costs associated with enforcing data-dependences and with task-related actions. The bottleneck analysis tool gives feedback to the programmers about data-dependences that limit parallelism. In particular, there are several “accidental dependences” that can be easily removed with large improvements in parallelism. These tools combine into a systematic methodology for performance tuning in Implicit Parallelization. Finally, armed with the criticality-driven model, the thesis revisits several architectural design decisions, and finds several encouraging ways forward to increase the scope of Implicit Parallelization.

To my parents.

ACKNOWLEDGMENTS

Graduate school has been an interesting journey. There have been frequent storms and encounters with rough seas with only brief periods of smooth sailing. Every time the ship seems to be under control, the sea throws up a new challenge. Often the ship has come close to sinking. But somehow I managed to make it through, enriched by the experience (not financially though). A big reason for my survival was the people with whom I shared parts or whole of the journey. I would like to express my gratitude to them here.

Firstly and foremost I would like to thank my advisor Matt Frank. He is at the same time one of the smartest people I have worked with yet one of the most humble. He always had the time and the patience to listen to me drone on even when it would be clear to him that I was off in la-la land. This was because his objective was to make my stay in graduate school a good learning experience rather than to maximize the utility he could get out of me, unlike several other professors. He would always come up with great insights and constructive feedback in each discussion. Discussions with him always leave me energized and with several new ideas to try out. He has inspired me to explore new areas and made doing research a memorable experience. At the same time he has been very understanding, never pushing me when I have been down in the dumps. I have learned a lot about doing good research from him.

Next I would like to thank my fellow members of the IPA group over the years. The IPA group at its peak boasted of an amazing team and I still marvel at the talent we had in our group. Kshitiz Malik has been my partner in crime during the time I learned the most. Collaborating with him has been a most fun experience. He played the role of devil's advocate to perfection whenever I pitched any new idea to him, ensuring that I had to think deep and hard before proposing any new theory and preventing me from doing "shotgun-style research". When he picks up a problem, he keeps working and obsessing over it until it is solved and the phrase "give up" doesn't exist in his

dictionary. His energy and enthusiasm have been a source of great inspiration to me.
Kevin Woley was the initial “architect” of the Polyflow architecture and wrote a large chunk of the simulator code. He designed and wrote high-quality code. He was always pushing technology to the edge by exploring new features, writing fun scripts to automate things, trying out and learning to use new tools. He made me appreciate the importance of learning new tricks and tools. My only regret being that I could not convince him to switch to vim from emacs (I almost did it though).
Sam Stone was one of the most hard-working and sincere people I have met. He planned things very well and made steady progress towards his goals, unlike other IPA members like me who slacked off until deadlines approached. I hope one day I can acquire some of those qualities.
Vikram Dhar was exploring and reading up new things and always gave interesting insights. Nitin Navale and Gene Wu made sure that my training would not be incomplete by making me watch the Star Wars series. They have made me a devout follower of Master Yoda. Indebted to them I am.
Nick Weaver always had strong opinions about issues and it was very interesting to debate with him, especially about American politics.
I would also like to thank other members of the architecture group at UIUC. Pradeep Ramachandran, Naveen Neelakantam, Pierre Salverda, John Kelm, Aqeel Mahesri and several others were kind enough to review my papers and attend practice talks. They gave valuable feedback on my research and helped improve its quality. The architecture reading group was another forum where I learned a lot, although I was lazy and wasn’t as regular in attending it as I would have liked.
Steve Lumetta has been a very useful source of feedback on my research. He is someone who doesn’t tolerate any nonsense and is quick to point it out. He is also willing to debate for as long as it takes until the topic under discussion is convincingly resolved one way or the other. Discussing my research with him has forced me to frame my arguments precisely and several times has helped me iron out kinks in my theories or point out aspects that I have failed to consider. In addition, he cracks deep jokes and you have to push your mind to the limit just to understand them.
I would like to thank my committee members Sarita Adve, Pradeep Dubey, Josep Torrellas and Craig Zilles for their valuable guidance and feedback. They have been very kind with their time and have provided lot of encouragement.
I would like to thank all my friends who have provided support and helped make this journey

enjoyable.

Last, but not the least, I would like to thank my parents and my family. They have always supported me and have encouraged me in my pursuit of higher studies. I owe everything to them.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Motivation: The Need for Parallelization 1
1.2 Implicit Parallelization as a Potential Solution 2
1.3 Challenges to Implicit Parallelization 3
1.4 Contributions of this Thesis 4
1.5 Roadmap 5
1.5.1 Part I: Identifying and Quantifying Parallelism 6
1.5.2 Part II: Extracting Parallelism 6
1.5.3 Part III: Bottleneck Identification and Removal 7

PART I IDENTIFYING AND QUANTIFYING PARALLELISM 8

CHAPTER 2 AN UNDERSTANDING OF PARALLELISM 9
2.1 Heuristics for Parallelism 9
2.2 Dependence Height and Program Completion Time 11
2.3 Parallelism to Reduce Achieved Dependence Height 12
2.3.1 Techniques to Exploit Parallelism 13
2.3.2 Trade-Offs in Exploiting Parallelism 14
2.4 A Quantitative Approach to Parallelization 15

CHAPTER 3 CRITICAL PATH ANALYSIS OF PROGRAM EXECUTION 17
3.1 Lam's Abstract Model of Parallelism 17
3.2 Fields' Model of Superscalar Execution 19
3.2.1 Program Dependence Graph 19
3.2.2 Edge Latencies 21
3.2.3 Timestamp Assignment and Last-Arriving Edges 22
3.2.4 Program Critical Path 22
3.2.5 Slack and Tautness Analysis 23
3.3 Applications of Critical Path Analysis 24
3.3.1 Critical Path Analysis for Superscalar Processors 25
3.3.2 Critical Path Analysis for Parallel Systems 25
3.3.3 Critical Path Analysis for Speculative Multithreading 26

CHAPTER 4 PARALLELISM AND FETCH CRITICALITY 27
4.1 Fetch Criticality in Superscalar Execution 28
4.1.1 Methodology for Characterizing Critical Path 28
4.1.2 Prevalence of Fetch Criticality 31
4.1.3 Fetch Criticality Generating Events (FCGEs) 32

4.2 Fetch Criticality ⇒ Unexploited Parallelism . . . 34
4.3 Task-Based Parallelization to Alleviate Fetch-Criticality . . . 35
4.3.1 Control-Independent Task Spawning . . . 35
4.3.2 Dependence Graph Model for Control-Independent Task Spawn . . . 36
4.4 Necessary Conditions for Existence of Parallelism . . . 39
4.4.1 Rules for Successful Task Spawn . . . 39
4.4.2 Spawn Rules in Action . . . 39
4.4.3 Proof of Spawn Rules . . . 42

CHAPTER 5 QUANTIFYING PARALLELISM FROM POTENTIAL TASKS . . . 45
5.1 Task Benefit and Critical Path Length . . . 45
5.2 Assumptions About Impact of Tasks . . . 45
5.3 Estimating Task Benefit . . . 46
5.3.1 Definition: Adjusted Slack . . . 46
5.3.2 Performance Benefit from Spawning a Task . . . 47
5.3.3 Adjusted Slack Calculation for Synchronized E → E Edge . . . 48
5.3.4 Adjusted Slack for Spawn F → F Edge . . . 49
5.4 Overall Approach . . . 49
5.5 Validation . . . 50
5.5.1 Infrastructure and Methodology . . . 50
5.5.2 Validation Results . . . 50

PART II EXTRACTING PARALLELISM ON POLYFLOW . . . 64

CHAPTER 6 POLYFLOW: TARGET SPECULATIVE PARALLELIZATION SYSTEM . . . 65
6.1 Terminology and High-Level Overview . . . 65
6.2 Management of Data-Dependences . . . 67
6.2.1 Register Dependences . . . 67
6.2.2 Value-Prediction for Callee-Saved Register Dependences . . . 68
6.2.3 Memory Dependences . . . 68
6.3 Disambiguation of Memory Accesses and Forwarding of Data . . . 68
6.4 Non-Blocking Scheduling through Divert Queues . . . 69
6.5 Release Policy for Synchronized Instructions . . . 69
6.6 Task Spawn Management . . . 70

CHAPTER 7 RELATED WORK IN SPECULATIVE PARALLELIZATION . . . 71
7.1 Compiler-driven Automatic Parallelization . . . 71
7.2 Speculative Parallelization . . . 72
7.2.1 Challenges to Speculative Parallelization . . . 73
7.3 Task Selection for Speculative Parallelization . . . 73
7.3.1 Potential Task Choices Considered . . . 74
7.3.2 Heuristics to Estimate Task Benefit . . . 75
7.4 Program Transformations for Speculative Parallelizability . . . 77
7.4.1 Speculative Program Transformations . . . 77
7.4.2 Revisiting Application Implementation . . . 78

CHAPTER 8 TASK SELECTION FOR POLYFLOW 79
8.1 Comparison Policies 79
8.1.1 Closest Spawn Policy 79
8.1.2 Data-Dependence Count Policy 81
8.2 Task Selection in Polyflow 83
8.2.1 Impact of Threshold 83
8.2.2 Nesting Analysis for In-order Task Spawning 84
8.3 Understanding Performance 87

PART III ENHANCING PARALLELISM THROUGH BOTTLENECK REMOVAL . 89

CHAPTER 9 APPLICATION BOTTLENECKS TO PARALLELIZATION 90
9.1 Background 90
9.1.1 Abstract Dependence Height Analysis 90
9.1.2 Critical Path Analysis 91
9.2 Design of SPARTAN 91
9.2.1 Functionality 92
9.2.2 Bottleneck Identification 92
9.2.3 Bottleneck Quantification 93
9.3 Bottleneck Analysis for Benchmarks 93
9.3.1 Bottlenecks in VPR Place 94
9.3.2 Bottlenecks in Twolf 95
9.3.3 Bottlenecks in Parser 96
9.3.4 Bottlenecks in Gzip 96
9.3.5 Discussion 97
9.4 Quantifying Bottlenecks and Validation 97
9.4.1 Quantifying Bottlenecks in VPR 97
9.4.2 Potential for Parallel Performance on Polyflow 98
9.4.3 Speculative Parallelization of VPR 99

CHAPTER 10 ARCHITECTURAL BOTTLENECKS TO PARALLELIZATION . . 100
10.1 An Upside Potential Study 100
10.1.1 Methodology 100
10.1.2 Architectural Constraints Modeled 101
10.1.3 Idealizations in the Study 102
10.1.4 Results 102
10.2 Task Granularity and Parallelism 105
10.2.1 Results and Analysis 107
10.2.2 Task Granularity in Swim 108
10.3 Cost of Enforcing Inter-Task Data Dependences and Task Penalties 109
10.4 Nested Parallelism and Out-of-Order Task Spawning 111
10.5 Impact of Constraining Available Cores 113

CHAPTER 11 CONCLUSIONS 115
11.1 Thesis Summary 115

REFERENCES 117

AUTHOR'S BIOGRAPHY 123

CHAPTER 1

INTRODUCTION

1.1 Motivation: The Need for Parallelization

The microprocessor industry is entering a new era. For the past decade or so, computer architects have been able to successfully convert the additional transistors made available by Moore's law into useful performance. This has been achieved through wider pipelines, out-of-order execution, and aggressive speculation accross branches. However, these techniques face severe roadblocks to providing further improvements in performance. The roadblock to scaling these techniques arise in the form of higher power requirements, diminishing improvements in performance, and circuit complexities. Some have referred to these barriers as the power wall, the ILP (Instruction Level Parallelism) wall, and the complexity wall [1].

In response, the industry has made an about turn over the last few years and is now moving towards multicore architectures comprising simpler rather than more complex cores [2]. The hope is that applications can profitably take advantage of parallelism to utilize extra cores for improvements in performance. However, expliticly parallelizing applications is a hard problem and places a huge burden on the programmers. It requires programmers to make a judgement as to where parallelism lies in their applications such that it can be profitably be utilized to deliver performance. Next, programmers need to correctly parallelize their applications to obtain an equivalent parallel application. This can be complicated by unanticipated data-dependences that might exist between seemingly unrelated regions of the program, not enforcing which might introduce subtle errors. Finally, performance debugging a parallel code is also challenging since new issues like false sharing, lock contention, load imbalance, overhead of tasking, etc. might swamp any benefits of parallelism. All of these can be hard challenges especially for large applications where a single programmer might not have an understanding of the entire code base.

The responsibility of manually parallelizing applications is at odds with the current approach that has served programmers well for a long time. The traditional approach for application development most commonly used has been to write sequential applications and rely on the architecture to find instruction-level parallelism “under-the-covers”, while externally presenting sequential semantics. Moore’s law meant that with each new generation, microprocessors got faster and could exploit more ILP. Thus, the same program used to automatically perform better with each new generation of processors.

Programmers would like the trend of automatic scaling of performance with new generations of microprocessors to continue. Such a separation of concerns keeps programming simple and allows it to be accessible to a wide audience. At the same time, it enables programmers to achieve high performance on their applications without having to worry about architecture-related most of the times. The multicore era threatens to disrupt this tradition and cause a huge hit to programmer productivity. In other words, there is a disconnect between the requirements on the programmer side and trends in the architecture side.

1.2 Implicit Parallelization as a Potential Solution

This thesis explores a solution to the above-mentioned disconnect that would allow programmers to continue writing applications following the sequential programming model, but reap the benefits of additional cores through under-the-covers parallelization. This thesis refers to such an approach as “Implicit Parallelization”. Figure 1.1 illustrates the objective of Implicit Parallelization.

This approach offers the promise of maintaining programmer productivity by allowing programmers to keep writing single-threaded applications. At the same time, it would deliver high performance by automatically and implicitly parallelizing the application into tasks that can execute concurrently on the available cores. Thus it would continue the hugely successful tradition of hiding parallelization details from programmers. This parallelization is to be carried out while preserving the sequential semantics, so that correctness and ease of debugging would not be compromised. Finally, the parallelization system can automatically tailor the parallelization for characteristics of the underlying architecture (which can vary quite a bit). This frees up the programmer from worrying about architecture-specific issues like granularity of parallelism.

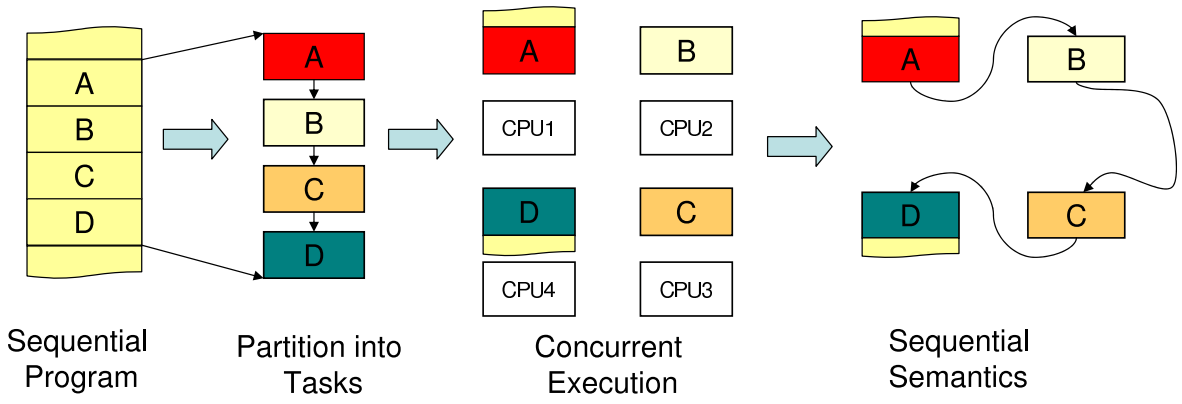

Figure 1.1: The high-level objective of Implicit Parallelization.

1.3 Challenges to Implicit Parallelization

Implicit Parallelization is not without its challenges. Tthe approach has previously been explored by several researchers and for several years under various guises such as automatic parallelization and speculative parallelization. Limit studies suggest huge potential for such approaches. However, most previous research prototypes have demonstrated modest potential on sequential benchmarks while adding significant complexity to hardware (and potentially middleware such as compilers, etc). This is because implicit parallelization on multicore architectures is a much harder problem than under-the-cover parallelization for Instruction-level Parallelism (ILP) on superscalar architectures. There are challenges at multiple layers: at the architecture-level as well as the application-level.

At the application level, the parallelization needs to be carried out while preserving application semantics. This requires respecting application control- and data-dependences while parallelizing it. Enforcing these dependences accross tasks running on different cores typically introduces a large amount of cost to the parallelization process. This cost can have a huge bearing on the performance of Implicit Parallelization approaches. Therefore, applications should be partitioned into tasks that yield high performance in spite of these costs. This is hard because static dependence analysis is hard. Offline tools might not even know of the existence of all dependences. Even if dependences are known, the costs associated with dependences can vary and depend upon many dynamic effects such as cache misses and mispredicted branches. Optimal

partitioning into tasks is a hard problem, and good heuristics are needed.

In addition, there are significant challenges at the level of the architecture. Modifications have to be made to the base multicore architecture. These are needed to perform task spawns and merges internally, enforce inter-task data-dependences, manage inter-task data communication, and ensure correct execution semantics. Trade-offs need to be made between hardware complexity and performance.

1.4 Contributions of this Thesis

This thesis approaches challenges in Implicit Parallelization from a new direction, by casting the problem of parallelization in terms of instruction criticality. This new approach enables quantification of trade-offs previously understood qualitatively, and naturally unifies several previously used heuristics for finding parallelism into a single framework. This allows the thesis to to improve upon the performance of previous attempts in several aspects of the problem.

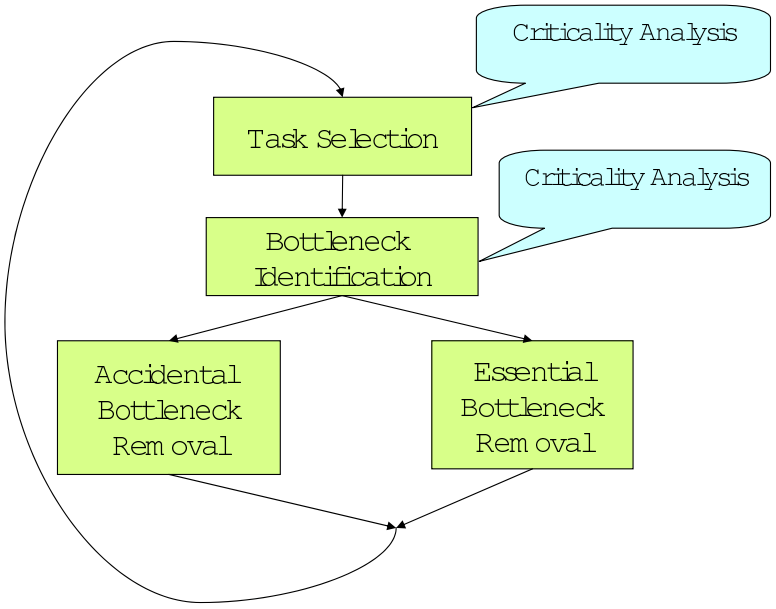

Figure 1.2: An application-centric Implicit Parallelization flow.

Building upon this new approach, this thesis develops a flow for Implicit Parallelization of programs. This flow develop a systematic methodology for obtaining high performance through

Implicit Parallelization of programs. The thesis develops tools for parallelizing applications as well as for performance debugging that form key components of this flow. A programmer for whom the initial best-effort parallelization carried out by the system doesn't perform sufficiently can use these tools and iterate over the flow to tune the application until performance goals are met. The flow is illustrated in figure 1.2.

As mentioned above, the key insight in this thesis is the application of concepts of instruction criticality and critical path analysis to the parallelization problem. This is used to develop a criticality-driven model of implicit parallelization. The model has several applications. It can be used to make quantitative predictions about the parallelism exposed from different task choices. It is a key ingredient of the task-selection phase. A task selection policy developed based upon this model significantly outperforms previous work in this area.

The criticality-based model is also used in the application bottleneck identification phase of the flow. This thesis finds that the application bottlenecks to parallelism in form of data-dependences fall into two categories: accidental dependences and essential dependences. Accidental dependences can be easily removed through standard transformations such as reassociation, or through calls to better (more parallelizable) library functions. Removing essential dependences, on the other hand, requires rethinking the algorithm and potentially trading-off output quality with performance.

In addition to the above flow, this thesis also revisits the architectural decisions made by most Implicit Parallelization systems. This includes decisions about the granularity at which parallelism will be extracted, techniques to manage data-dependences, and the impact of limiting cores. Results shows encouraging directions to move forward.

1.5 Roadmap

This thesis deals with the challenge of Implicit Parallelization of applications. As described above, a one-step approach is unlikely to always lead to success. Rather, a systematic flow is developed that allows iterative refinement of the application until performance goals are met. The thesis is broadly organized along the lines of the parallelization flow described.

1.5.1 Part I: Identifying and Quantifying Parallelism

The first part of the thesis deals with the problem of identifying parallelism in applications. Chapter 2 defines parallelism in terms of a dependence graph in a way that naturally unifies several different heuristics for parallelism proposed in literature (such as Memory-Level Parallelism or MLP, Branch-Level Parallelism or BLP, Data-Level Parallelism or DLP, etc.). Chapter 3 describes two types of dependence graphs described in literature. These can be used to represent program execution and for finding the program critical path. The chapter also describes some applications of critical path analysis from literature.

Chapter 4 develops the treatment of parallelism further for task-based parallelization by approaching it from an instruction-criticality perspective. It describes the relationship between potential for parallelism and fetch-criticality of instructions. In particular, whenever instructions are fetch-critical in a region, there might be potential for improving performance by spawning a task in that region provided a set of conditions are met. These conditions are developed into formal rules for the existence of exploitable parallelism. The chapter also describes how Fields' dependence graph model [3] for finding instruction criticality in superscalar execution can be extended for an Implicit Parallelization architecture.

The first part concludes with chapter 5 that develops the insights of chapter 4 into a quantitative model to predict the expected parallelism from spawning a potential task. The chapter validates the model by comparing the predictions from the model with the measurements from a prototype of a 4-core Implicit Parallelization system.

1.5.2 Part II: Extracting Parallelism

The second part of the thesis deals with the challenge of improving parallel performance by spawning tasks in an Implicit Parallelization system. The Implicit Parallelization system used in this thesis belongs to the class of "Speculative Parallelization (SP)" architectures because it can speculate on ambiguous data-dependences (typically memory-based) and recover if a speculation failed. The particular system used in this thesis is named "Polyflow". Polyflow has several features that differentiate it from other research SP systems. Chapter 6 describes the Polyflow architecture in some detail. Chapter 7 describes related work in Implicit Parallelization of applications,

including task-selection strategies used previously.

Chapter 8 describes the task selection strategy developed in this thesis. The strategy build upon the task benefit estimation model developed in chapter 5. This task selection strategy is compared to task selection policies used in previous work and is found to significantly outperform them because it is built on top of a better parallelism estimation model, and because it considers containment relationships between tasks in addition to individual task behaviors.

1.5.3 Part III: Bottleneck Identification and Removal

Finally, the third part of this thesis steps back and takes a look at the broader picture of Implicit Parallelization, both from the architecture side as well as for applications. Chapter 9 presents another application of the criticality model on the application side. It analyzes several benchmark applications and realizes that frequently, applications in their current form are not very amenable to parallelization. But a few tweaks can sometimes greatly enhance the scope of parallelization. In particular, there exist several "accidental dependences" which can be easily removed without causing much change to application behavior, but with great improvements in parallel performance. The chapter develops a tool, called SPARTAN, which can automatically find the important data-dependences that limit parallelism and also quantify the importance of each bottleneck.

Chapter 10 does the analysis in the architecture side. It finds that while Implicit Parallelization has high performance potential, some of the constraints imposed by current research prototypes severely limit the performance potential. Examples include the restrictions on task sizes that prevent exploitation of parallelism at large granularities. On the other hand, some of the other decisions like restricting task spawning to be in-order (as opposed to the more relaxed out-of-order spawning) don't matter as much in terms of performance while cutting down on complexity. This suggests encouraging ways forward to expand the scope of Implicit Parallelization.

Finally, chapter 11 draws conclusions and gives some final remarks on this work.

PART I

IDENTIFYING AND QUANTIFYING PARALLELISM

CHAPTER 2

AN UNDERSTANDING OF PARALLELISM

The microprocessor industry has recently moved away from the trend of higher performance through ever increasing clock speeds and wider pipelines and towards multiple (usually simpler and slower) cores. This has made it increasingly important for computer architects and software developers to deliver high performance by exploiting parallelism in applications. In response, researchers have explored various parallelism-enhancing architectural techniques and software transformations.

Several opportunities for parallelism have been identified and given different names. Section 2.1 summarizes some of these "heuristics" for finding parallelism. While these heuristics are useful to focus on specific opportunities, they can also end up limiting the scope of parallelization techniques since designers might target only a subset of these heuristics and miss out on other opportunities for parallelism. The ultimate goal is to improve performance, therefore parallelization techniques should target parallelism in general.

With this as motivation, sections 2.2 and 2.3 formalize a way of approaching parallelism that naturally unifies the different heuristics, in terms of the "dependence graph" representation of program execution. Parallelization techniques also typically create a performance trade-off since there are costs involved to exploiting parallelism. Section 2.4 describes how the above treatment of parallelism can be used to evaluate this trade-off quantitatively to design parallelization techniques and policies.

2.1 Heuristics for Parallelism

Parallelism might be exploited at a fine granularity, as low as the level of a few individual instructions. This is commonly referred to as instruction-level parallelism (ILP). Examples of ILP

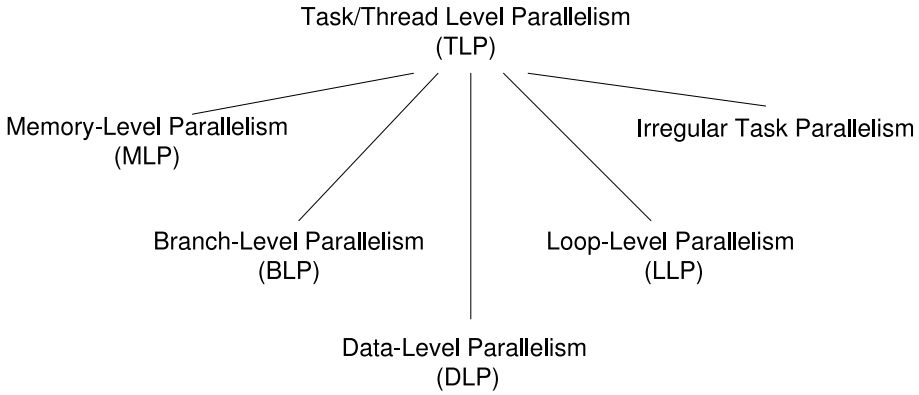

Figure 2.1: Some heuristics proposed for finding task level parallelism.

techniques are multiple (potentially duplicate) functional units and out-of-order execution. Multiple functional units allow several independent instructions to execute simultaneously rather than wait on just one execution unit. Out-of-order execution allows simultaneous execution of multiple non-consecutive independent instructions within a fixed-sized window even when some of the intermediate instructions might be waiting for their producers to complete.

Parallelism can also be exploited at coarser granularity by dividing up the sequential execution of the program into chunks of instructions, and by (partly or completely) overlapping the execution of these chunks, commonly referred to as “tasks” or “threads”. These tasks can run in parallel on multiple processors of a multiprocessor or a multicore architecture, or different hardware threads of a multithreaded processor. These tasks can help achieve higher throughput than possible with a single stream of execution. This form of parallelism is the primary focus of this thesis, and will be referred to as Task-Level Parallelism (TLP) in this thesis.

Several opportunities for TLP have been identified in literature. Terms such as data-level parallelism (DLP) have been coined. Data-level parallelism occurs mostly in loops where the same piece of code executes on a large amount of data. Other irregular task-level parallelism is said to exist where a sequential execution can be separated into multiple tasks that can execute concurrently. Note that these tasks might not be completely independent, in which case the inter-task dependences must be enforced for correctness.

Other opportunities for TLP are created by dynamic events such as instruction and data cache misses and mispredicted branches, which cause single execution streams to achieve only a small fraction of the peak throughput allowed by the processor. Heuristics to spawn tasks to take advantage of memory-level parallelism (MLP) [4] and branch-level parallelism (BLP) [5] have

been proposed to improve performance when these events are prevalent. Further, parallelism might occur in regular structures like loops, as well as in irregular forms.

2.2 Dependence Height and Program Completion Time

Another way of approaching parallelism is by thinking of program execution in terms of a "dependence graph". Given an application program that executes on a given input data and a processor architecture, there are a set of application dependences that must be enforced. These are the control-dependences (from a branch to instructions control-dependent on the branch) and data-dependences (from an instruction that produces data to instructions that consume the data) enforced by the architecture. Note that for a particular kind of architecture, these enforced dependences may be a subset or a superset of the "true" control- and data-dependences. For example, with control speculation and value prediction techniques, some of the true control- and data-dependences can be removed. On the other hand, some "false" dependences such as anti and output data dependences, and extra control dependences might be enforced by the architecture. These enforced control- and data-dependences constrain the earliest time by which the program can possibly hope to complete. This earliest time is determined by the longest chain of dependences (both control and data) in the program trace, by taking the sum of the minimum latency (again a function of the type of architecture) required for each operation in the dependence chain. It is referred to as *dependence height* of the program.

However, an actual processor implementation enforces several other constraints (dependences) besides the application control- and data-dependence constraints. These constraints come from limited resources, bandwidth constraints, and other architectural restrictions on program execution. Examples are limited buffer sizes, in-order fetch and retirement of instructions (due to Von-Neumann design), possibly in-order execution, etc. In addition, instructions might take much longer to execute than their minimum required latencies. This might be due to limited bandwidth, limited cache sizes, etc.

Constraints such as those described above cause the application completion time to typically be much longer than the dependence height. That is, *the achieved dependence height* on an actual processor can be much longer than the *minimum dependence height* due to architectural, resource

and bandwidth constraints [6]. Such additional constraints (dependences) delay instructions that could have otherwise completed much earlier (when constrained only by control- and data-dependences). Also the achieved dependence height might be longer than the minimum dependence height because instructions might take longer to execute than their minimum required latency (cache misses, etc).

The execution of the program then can be visualized as a graph where nodes can represent individual operations and directed edges represent dependence relationship between two operations. For example, Lam et. al [7] represented the execution of each instruction for a given program trace as an individual node, and edges between nodes capture the true control- and data-dependence constraints. Edges are labelled with the minimum latency for that operation. The dependence height can be computed by finding the longest chain of dependent nodes where the length of a chain is given by the sum of edge latencies on that chain. Fields et. al [3] describe a graph representation that can model, in addition to the true control and data dependence constraints, several other architecture-imposed constraints as well. They refer to the longest dependence chain as the "critical path" of program execution. Further details of these dependence graphs will be presented in chapter 3. The next section gives a definition of parallelism in terms of such a "dependence graph" structure.

2.3 Parallelism to Reduce Achieved Dependence Height

Speeding up program execution from its current performance on a given processor architecture requires reducing the achieved dependence height. There can be two ways to achieve this reduction:

- Reduce the latency for some of the operations on the longest dependence chain.
- Break some of the edges on the longest dependence chain so that the resulting longest dependence chain is shorter in length.

The first category captures techniques like higher clock speed for the processor and/or memory, larger caches (to convert previous misses into hits thereby reducing latency), etc. It can also include techniques like software/hardware prefetches. These techniques keep the structure of the

dependence graph intact (or almost intact), but still manage to reduce the dependence height (and thereby the program completion time) by cutting down on the latency of some of the operations, in particular, some of the operations on the longest dependence chain.

The second category of the techniques speed up program execution by removing some of the constraints (edges) were previously sequentializing operations on the longest dependence chain. These techniques therefore allow these operations to start executing concurrently where previously they had to execute sequentially, thus reducing the length of the previously longest dependence chain. These techniques, therefore, exploit the potential for "parallelism" in the application. Broadly, parallelism refers to the ability to reduce the achieved dependence height by removing some of the previously imposed sequentializing constraints (dependences) on execution. This removal of constraints allows the architecture to perform previously dependent (or transitively dependent) operations in "parallel", that is concurrently. Since a given set of operations have to be completed to complete the overall program, parallelism allows these operations to be finished earlier than before by allowing concurrent execution. Exploiting parallelism therefore can lead to a increase in performance by reducing the achieved dependence height.

2.3.1 Techniques to Exploit Parallelism

Opportunities for parallelism can be created in the following ways (among others):

- Program transformations to remove previously enforced control- and data-dependences while preserving semantic behavior.
- Architectural designs that remove/modify some previously enforced constraints that caused sequentialization.
- As a special case, additional bandwidth and resources that allow instructions waiting on resources to start execution earlier than before. Or else, smarter resource allocation policies that make better use of the available resources.

Several program transformations can achieve the effect of removing control and data dependences for parallelism. Loop unrolling removes the control-dependence upon loop-branch for several instructions. Techniques like renaming, privatization, etc. can remove previously imposed

data-dependences. Explicit parallelization into threads/tasks can remove control-dependences (and possibly data-dependences) that previously sequentialized different regions of the program, to exploit “Task-Level Parallelism (TLP)”. If the threads/tasks are iterations of a loop operating on the same code but on different data values, this has been referred to as “Data-Level Parallelism (DLP)”.

Architectural design techniques can relax some of the dependences that limit the achieved dependence height. Out-of-order execution removes the constraint of sequential issuing of instructions (i.e. a dependence from each instruction’s issue to that of the next one), and takes advantage of “Instruction-Level Parallelism (ILP)” in the program. Control-speculation for correctly predicted branches can remove previously imposed control-dependences due to those branches. Similarly, correct value prediction can remove previously imposed data-dependences, as can techniques like renaming of registers and memory. Other architectural techniques have been proposed to overlap the penalty of cache misses by breaking dependences that sequentialize them, and have been placed under the category of “Memory-Level Parallelism (MLP)” techniques. Larger number of resources can also reduce the achieved dependence height because limited resources can create dependences between otherwise unrelated operations. For example, a larger scheduler means that otherwise ready instructions that were delayed because they could not find a slot in the scheduler can now start execution earlier.

2.3.2 Trade-Offs in Exploiting Parallelism

Several architectural and software techniques have been proposed to exploit parallelism in applications for high performance. However, these techniques can also introduce extra costs in other aspects of a program’s execution that did not exist earlier. The costs might be due to:

- The extra hardware resources required by the architectural technique, which might be associated with extra power consumption, area requirements, design complexity, etc. In addition, these might lead to increased clock cycle time which could increase dependence height by slowing down some (or maybe all) operations on the longest dependence chain.
- Architectural parallelization techniques might relax/remove some dependence constraints but introduce new ones. In addition these might also increase the latency of some operations.

For example, control speculation can remove control-dependences when successful, but can add a misspeculation penalty to the fetch time of the correct branch target when it fails.

- Software parallelization techniques might create new cost in terms of additional operations or introduce new control and data-dependences.

The costs associated with a parallelization technique therefore introduce a performance trade-off that governs the profitability of the technique in improving performance. The trade-off might be hard to reason about qualitatively, since the benefits and costs can vary depending upon application behavior. Qualitative approaches are usually required to judge the value of a proposed parallelization technique. Further, some parallelization techniques can be selectively applied only to a few chosen regions of the program. In such cases, a policy is needed to decide where the technique should be applied. Success depends upon the ability of the policy to incorporate the parallelization cost-benefit trade-offs in its decision process.

2.4 A Quantitative Approach to Parallelization

In terms of a dependence graph, a parallelization technique can reduce the dependence height by breaking some dependence constraints that previously existed on the longest dependence chain. In other words, parallelization can potentially reduce the execution time by removing some of the edges on the program critical path (since critical path is simply the length of the longest dependence chain). On the other hand, a parallelization technique can also add new nodes and edges elsewhere in the graph as well as increase some edge latencies, and the resulting critical path might turn out to be worse off than the original one. Therefore parallelization techniques should be applied when the performance trade-off is in its favor.

The treatment of parallelism developed in this chapter can be used as a quantitative approach to evaluate the performance trade-off for a proposed parallelization technique to decide if it could be worthwhile, as well as to design decision policies to decide where it is most profitable to parallelize. The approach would be to estimate the impact of the parallelization on the height of the dependence graph (i.e. length of the program critical path), and use this to decide if the performance trade-off favors parallelization.

The advantages of this approach can be manifold. First, it can capture all “forms” of parallelism since it doesn’t differentiate between them. Second, with a well-chosen dependence graph representation, this analysis can be much quicker but still quite accurate compared to actually prototyping the parallelization technique. Finally, if a policy is required to decide how to parallelize, there can be a large space of choices available, and evaluating each choice by actually parallelizing can be computationally expensive.

The following chapters will show, for the case of implicit task parallelization, how this approach allows a quick and accurate exploration of this space. The quick exploration is made possible because parallelization perturbs the dependence graph in only a few places (few edges added/removed/modified) and the rest of the graph remains unaffected. This enables a very quick estimation of the reduction in dependence height (or critical path length) because of the proposed parallelization. The accurate exploration is possible because this approach finds parallelism “in general” rather than being limited to specific heuristics.

CHAPTER 3

CRITICAL PATH ANALYSIS OF PROGRAM EXECUTION

Chapter 2 described a quantitative approach to parallelism. One of the requirements to approach parallelism quantitatively is a model of program execution that can capture different constraints under which the execution proceeds: both application-level and architecture-level. This chapter describes some of the models that have been developed in literature and their use in understanding and designing parallelization techniques.

Section 3.1 describes the abstract model used by Lam et. al [7] to explore the impact of different techniques to handle control flow on parallelism. Lam's model is quite optimistic because it incorporates the effect of only application control- and data-dependences. However, parallelism is affected by architectural factors as well and Lam's abstract model fails to capture those constraints. Section 3.2 describes Fields' model [3] of program execution. The model is more detailed than Lam's model. In addition to control- and data-dependences, it can capture several other type of architectural dependences that constrain program execution on superscalar architectures. A critical path analysis of program execution using this model can provide valuable insights about the bottlenecks to performance. Critical path analysis in one form or another has been used for analyzing and designing parallelization techniques in several systems. Section 3.3 describes some applications of critical path analysis from literature.

3.1 Lam's Abstract Model of Parallelism

Lam et. al [7] did a limit study on traces of several benchmark applications to understand how different ways of handling control flow in applications impacts the achievable parallelism. The study was motivated by the huge disparity reported between limit studies for aggressive out-of-order superscalar processors that speculated across branches such as the one conducted by

Wall [6], and the upside potentials for dataflow height studies assuming no constraints from control flow on performance (i.e. perfect branch prediction).

Lam's study started with a naive base model that imposed a control-dependence from a branch to all future instructions, and explored the impact of the following three improvements:

1. Speculation (SP): This removes true control-dependences from branches whose outcome can be predicted to future dependent instructions.

2. Control-dependence (CD): This removes the dependence from a branch to future control-independent instructions, thus freeing up instructions control-independent of a branch from having to wait for that branch's execution.

3. Multiple Flow (MF): This allows the ability to pursue multiple flows of control, and therefore multiple branches can be executed simultaneously (as allowed by CD and SP constraints).

The study evaluated several models of execution that combined the above techniques in different ways. Examples were base, SP, SP+CD, SP+CD+MF, CD+MF, etc. In order to estimate the impact of these techniques on parallelism, the study constructed a dependence graph representation of program execution for each model of execution considered. Nodes in the graph represented execution of individual instructions. Edges between nodes represented two kinds of dependences:

- True data-dependences in the program: Only true producer - consumer data dependences were represented. Anti- and output- dependences were eliminated (both register and memory) to capture the impact of renaming techniques. In addition, some true dependences were also removed to account for compiler optimizations. These included dependences such as those from stack pointer updates, loop index and induction variable updates.

- Control dependences: Call and return dependences were removed in all models to account for inlining transformations. The machine models differed in the control dependences enforced as described above.

Each edge in the dependence graph was labelled with a unit latency. The study built a dependence graph for each combination of the three techniques for handling control-flow. For each such graph

built, the dependence height gave the length of the longest chain of dependences in the graph, and therefore the minimum completion time for the abstract model represented by the graph. While the study didn't use the term, the longest dependence chain is basically the "critical path" of program execution on the machine model. The study found that the SP-CD-MF point enabled orders of magnitudes higher amounts of parallelism than possible in an aggressive superscalar processor (the SP configuration). The SP-CD-MF represents an optimistic upside potential of parallelizing the application in its given form on a multicore architecture.

3.2 Fields' Model of Superscalar Execution

Lam's study is a good quantitative approach to evaluating the impact of different techniques for handling control flow on parallelism. However the model is too abstract and focusses only on control and data dependences. It doesn't incorporate the effect of architectural constraints which have a large role in determining performance. Fields et. al [3] developed a dependence graph representation for program execution on superscalar processors that captures several architectural constraints in addition to control and data dependences. The model is described here.

3.2.1 Program Dependence Graph

Fields, Rubin, Bodík (FRB) developed a dependence graph [3] that can represent the constraints imposed by a superscalar architecture on the execution of an application trace. The dependence graph is a directed graph induced on the trace of committed program instructions. Note that the trace contains only instructions that are eventually committed, so the incorrectly fetched (or squashed) instructions are not included in the trace.

Each instruction is represented by three nodes, to capture the flow of the instruction through various stages of the superscalar pipeline. The first node (labeled "F") represents, in addition to *fetch* of the instruction, its decode, address generation, renaming and dispatch. The "E" node represents (out-of-order) issue and *execution* of the instruction. The "C" node represents instruction *commit*.

Graph edges represent dependences/constraints on execution. Table 3.1 summarizes the different dependences enforced in the model. Figure 3.1 illustrates the different types of dependence edges.

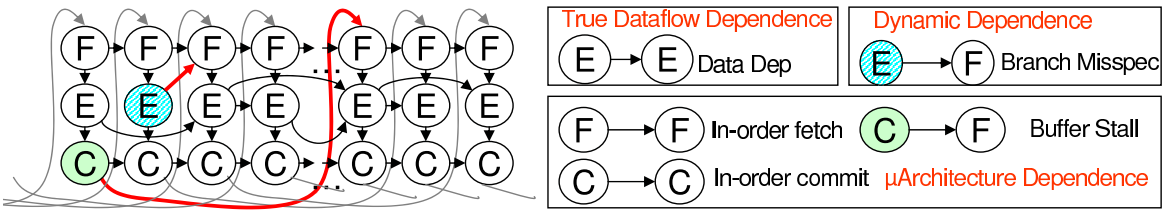

Figure 3.1: Explanation of dependence edges in FRB representation of program execution.

Name	Constraint modeled	Edge	Comment
FF	In-order fetch	$F_{i-1} \rightarrow F_i$	Instruction i cannot fetch before $i-1$.
EF	Failed Speculation	$E_{i-1} \rightarrow F_i$	Instr $i-1$ is a misspeculating instruction (mispredicting branch/load, etc).
CF	Finite reorder buffer size	$C_{i-r} \rightarrow F_i$	Instr i cannot fetch before instr $i-r$ commits, r is the size of the reorder buffer
FE	Execution follows fetch	$F_i \rightarrow E_i$	An instr cannot execute before it has fetched.
EE	Data dependences	$E_j \rightarrow E_i$	Instr j produces an operand of i.
EC	Retire follows execution	$E_i \rightarrow C_i$	An instr cannot retire before execution.
CC	In-order retirement	$C_{i-1} \rightarrow C_i$	Instr i cannot retire before $i-1$.

Table 3.1: Edges in the superscalar Program Dependence Graph.

Data-dependences are captured through EE edges, from the E node of producer to those of consumer instructions. These can include true data-dependences as well as other architecture-imposed data-dependences (such as anti- or output-dependences if the architecture doesn't rename instructions).

Several edges model microarchitectural constraints. For an instruction, fetch precedes execution, which in turn happens before commit. Thus, within each instruction, there is a FE edge, and an EC edge. Additionally, in a superscalar processor, all instructions are fetched in-order, so a FF edge flows between successive instructions. Likewise, in-order retirement of instructions leads to a CC edge from an instruction to the subsequent instruction. The processor's reorder buffer contains only N instructions so the processor must stall the fetch unit whenever there are more than N uncommitted instructions. Thus there is a CF edge from each instruction to the Nth succeeding instruction in the trace.

Enforced control-dependences are represented by a EF edge from the E node of the branch to the F node of the succeeding instruction, and therefore transitively to all future instructions, since

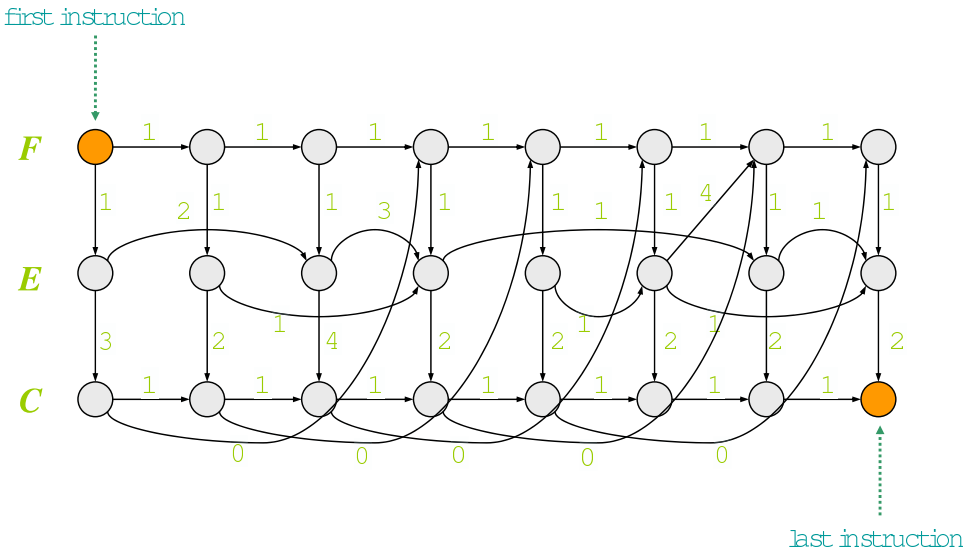

Figure 3.2: An example of a Program Dependence Graph for superscalar execution.

there is an FF dependences between all successive instructions. For processors that speculate past branches, this EF dependence exists only from the E node of the mispredicted branch to the F node of the succeeding instruction, representing the correct target. This is because the execution of a mispredicted branch causes the machine to roll back state, and restart fetching from the correct target. A similar EF edge can represent other misspeculation events as well, such as memory dependence violation due to out-of-order execution of loads with respect to producer stores.

3.2.2 Edge Latencies

Each graph edge is labeled with the latency induced by the dependence. This captures the minimum latency that a dependent operation has to wait after this operation has started. In addition, the edge latency captures the impact of resource contention as well. So, for example, EE edges are labeled with the instruction's latency through the functional unit (FU) as well as the time that the instruction had to wait to issue because the required FU was not available and was allocated to other instructions.

Branch misprediction (EF) edges from the Execute nodes of the branch to the Fetch node of the succeeding instruction are labeled with the number of cycles between the branch waking up and the fetch unit being restarted at the correct target. This can be quite large for deep pipelines [8] since a large number of instructions in the intermediate pipeline stages might need to be squashed

and there can be a long delay to warm up the pipeline.

Edges may also be labeled with a 0 latency. For example, some machines can fetch multiple instructions in a single cycle. The Fetch to Fetch edges between instructions fetched in the same cycle are labeled with 0 latency, while Fetch to Fetch edges from the last instruction fetched in a cycle to the first instruction fetched in the next cycle are labeled with a 1-cycle latency. Figure 3.2 shows an example dependence graph with labeled edges.

3.2.3 Timestamp Assignment and Last-Arriving Edges

Given the dependence graph with edges labelled by latencies as described above, each node in the graph can be assigned a timestamp. The timestamp represents the earliest time when the incoming dependences on the node allow the node's execution to proceed. This can be done using Wall's efficient algorithm for trace-based microarchitectural simulation [6]. For each incoming edge at a node, an "arrival" timestamp can be associated with that edge by taking the time associated with the producer node and adding the assigned edge weight. This represents the earliest time that the particular incoming dependence upon the node could have been satisfied given the modeled constraints. The timestamp associated with a node is then the maximum of the times calculated for all of its incoming dependence edges. This represents the idea that each node of each instruction may not start its action until all of its dependences are satisfied.

In particular, the incoming edge with the largest associated timestamp is called the *last-arriving edge*. If two edges arrive at a particular node at the same time we arbitrarily choose one of them as the last-arriving edge. Note that the graph of last-arriving edges is fully connected, contains every node, and forms a tree. This is because the graph is acyclic and each node has as a parent the predecessor node that produced the last-arriving edge. The path through the tree from the start node to any particular descendent represents the longest path to that node.

3.2.4 Program Critical Path

The dependences enforced by an architecture decide the program running time. In particular, the longest chain of dependence edges in the graph (when weighted by edge latencies) represents the earliest possible completion time of the program on that architecture, and is also referred to as the

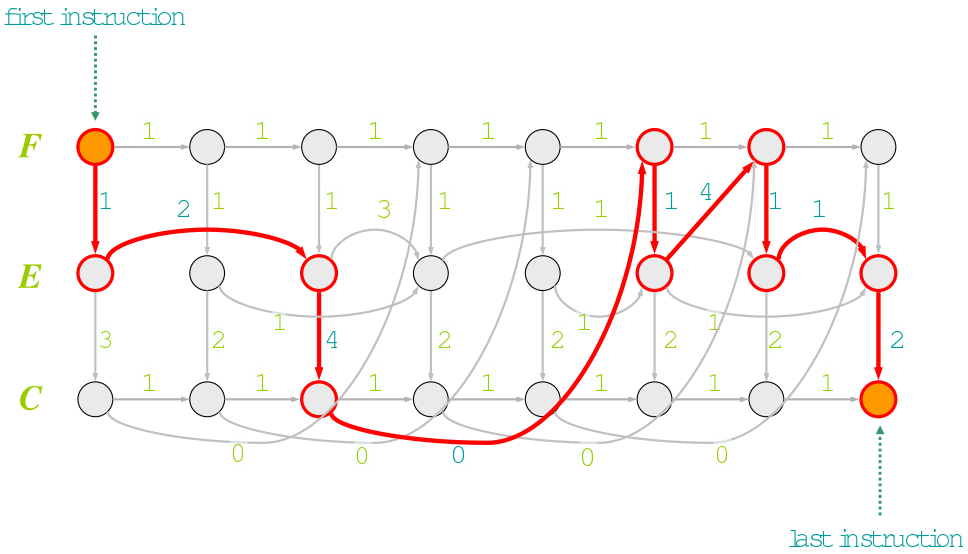

Figure 3.3: A Superscalar Program Dependence Graph with critical path highlighted.

critical path of program execution. Because of the structure of this graph, the critical path is guaranteed to flow from the F node of the first instruction in the trace to the C node of the last instruction. Therefore, *the longest path from the Fetch node of the first instruction to the Commit node of the last instruction represents the* **critical path** *of the program [3].* One the timestamp assignment has been done, the critical path can be easily found by following the last-arriving edge from the C node of the last instruction until the F node of the first instruction is reached. Figure 3.3 illustrates the critical path for the dependence graph of figure 3.2.

An instruction is said to be *critical* if any of its three nodes is on the critical path through the program. An instruction is termed *fetch-critical* if its fetch node is on the critical path. An instruction is *execute-critical* if its execute node is on the critical path (but not fetch node). An instruction is *commit-critical* if only its commit node is on the critical path.

3.2.5 Slack and Tautness Analysis

Only a subset of the nodes and edges in the dependence graph lie on the critical path. For the remaining nodes and edges, a useful characterization is the amount of slack on them. *The slack on a dependence edge is the number of cycles by which the edge can be delayed without affecting the program completion time.* The slack on a node in the graph can be defined similarly. Note that by definition, the nodes and edges on the critical have no slack.

Tune et al. [9] proposed a metric called *tautness*. The tautness of an instruction is the maximum number of cycles that the execution time might be reduced by executing that instruction earlier. Tautness can be similar defined for a dependence edge. This is a useful measurement because it quantifies the maximum payoff (in cycles removed from the execution time of the program) of applying an optimization to an instruction. It is a good measure the dominance of the critical path, if there is a large amount of tautness on a critical edge, it means that the next longest path (that doesn't include the edge) is at a large distance from the current critical path, and any parallelizing transformations that break/speed up this dependence will lead to a large improvement in performance.

3.3 Applications of Critical Path Analysis

Critical path analysis is a useful technique for bottleneck analysis. This thesis has been directly influenced to a large degree by the work of Fields et al. [3, 10, 11]. However, the area of critical path analysis is quite old and has been built on a large body of work. This section tries to summarize the development of critical path analysis, and give more depth on some of the relevant work in this direction.

The notion of critical path is useful for computing the minimum time required to complete a set of tasks in the presence of inter-task dependencies, and where each individual task requires a certain amount of time to complete. The set of tasks can be visualized through a graph with each node representing a task, a directed edge going from each task to the task that depends upon it and labelled with the latency equal to the time to complete the producer task. The critical path is the longest path in this graph. The length of the critical path gives the minimum amount of time required to complete the set of tasks, and each task on the critical path cannot be delayed without impacting the overall execution time. This concept was formalized in the 1950s [12] in the US Navy. A closely related concept is the notion of *slack*, which measures the amount of delay that can be tolerated by a task without affecting the overall completion time.

3.3.1 Critical Path Analysis for Superscalar Processors

The concept of critical path has been used in the area of computer architecture to understand program behavior on an underlying out-of-order superscalar processor and identify instructions that are critical to performance. Initial work focussed on long-latency load instructions, and heuristics to identify the loads that were critical to performance [13–15]. Calder et al. [16] used the longest data-dependence chain in the instruction window to approximate the critical path. Tune et al. [17] used heuristics such as monitoring unexecuted instructions at the head of reorder buffer to identify critical instructions. Fields et al. [3] showed how to find the critical path for superscalar execution as described before. In later work, Fields et al. used their dependence graph to measure instruction slack information [10], and interaction cost [18] that helps quantify the importance of different bottlenecks.

Critical path information has also found useful applications in superscalar processors to drive resource allocation decisions. It can be used to reduce the power consumption of instructions that are not on the critical path [19, 20], to direct non-critical instructions to slower functional units [10], and to drive steering decisions in clustered machines [21].

3.3.2 Critical Path Analysis for Parallel Systems

The notion of critical path has also been used to understand the execution of parallel programs on multiprocessor systems, both for shared-memory as well as message-passing systems. An early work on analyzing the critical path based on execution history of parallel programs was done by Yang et al. [22]. That study constructed a Program Activity Graph (PAG) to capture the program's execution. The graph represented computation within a parallel process, and communication between processes for send and receive operations. The critical path is computed to be the longest path in the PAG, and the study explored both centralized or distributed approaches to this computation.

However, efficient computation of critical path information is a challenging problem, since the size of PAG is proportional to program length. Hollingsworth [23] describes how to efficiently track the contribution of a set of specified procedures to the overall critical path length. The approach is to instrument communication events, as well as the events corresponding to entering and leaving of

specified procedures, and thus keep track of the longest path and procedure-specific statistics at each thread. This approach requires a low space overhead, leads to performance slowdowns in the range of 3-10 percent, and can be applied to message-passing and shared-memory programs. In addition, the study describes a technique called critical path zeroing, which bounds the improvement in performance from optimizing a given procedure.

Li et al. [24] adapt Fields' dependence graph for instruction execution on shared-memory multiprocessor systems built from in-order processors, and show how to compute the critical path and slack information from this graph.

3.3.3 Critical Path Analysis for Speculative Multithreading

Critical-path information has been used to drive policies in speculatively multi-threaded processors. Nagpal and Bhowmik [25] add latency to non-critical load instructions that might otherwise cause inter-thread data misspeculation. Tuck et al. [26] used a task level, rather than instruction level, dynamic criticality analysis to drive task scheduling for speculative multi-threading. Fields [11] pointed out that the dependence graph model could be used to identify good "cut-points" to partition a sequential application into multiple threads for parallel execution.

CHAPTER 4

PARALLELISM AND FETCH CRITICALITY

This chapter describes how to identify and quantify parallelism in applications. In particular, the focus is on parallelism that is not exploited by superscalar processors but can be extracted by spawning off a future region of the program as a task on a separate core of a multicore architecture. While that is the primary focus, the technique could potentially be extended to other ways of extracting parallelism. The approach developed here builds upon Fields' work on modeling superscalar execution described in chapter 3.

The chapter starts out in section 4.1 by analyzing the critical path of several benchmark programs for execution on a typical superscalar processor. A large number of instructions on the critical path are "fetch-critical", meaning that their fetch node was on the critical path. This phenomenon occurs because while superscalar processors can exploit parallelism in a limited window, they leave large amounts of parallelism in distant regions unexploited. In particular, there are three class of events that cause fetch-criticality in superscalar execution, and these are termed as Fetch-Criticality Generating Events (FCGEs).

The unexploited distant parallelism in superscalar processors manifests itself as fetch-critical instructions on the critical path. This is an important insight that can make it possible to identify and quantify the potential for exploiting distant parallelism through task spawning. This connection between fetch-criticality and parallelism is drawn in section 4.2. One way to overcome the restrictions on in-order fetch placed by superscalar processors is to spawn tasks on a separate core which allows a distant region to be fetched out-of-order. Section 4.3 extends Fields' dependence graph model for the scenario where tasks are spawned to different cores in a multicore architecture. Finally, section 4.4 describes the necessary conditions for existence of exploitable parallelism through task spawns. These conditions are formulated in terms of the dependence graph model of task spawning developed in section 4.3.

Parameter	Value
Pipeline Width	4 instrs/cycle Multiple taken branches per cycle
Branch Predictor	8K-entry Combined, 8K-entry gshare, 8K-entry bimodal, 8K-entry selector, 13 bits of history
Misprediction Penalty	8 cycles
Reorder Buffer	128 entries
Scheduler	128 entries
Functional Units	4 identical general purpose units
L1 I-Cache	32Kbytes, 4-way set assoc., 128 byte lines, 10 cycle miss
L1 D-Cache	32Kbytes, 4-way set assoc., 64 byte lines, 10 cycle miss
L2 Cache	512Kbytes, 8-way set assoc., 128 byte lines, 200 cycle miss
Memory Dependence Predictor	Ideal

Table 4.1: Superscalar parameters used for criticality characterization experiments.

4.1 Fetch Criticality in Superscalar Execution

4.1.1 Methodology for Characterizing Critical Path

Analysis of the program critical path can be a useful tool to gain insights about the application characteristics, as well as how the underlying architecture constrains the achieved performance. Table 4.1 describes the superscalar processor used for the studies in this section. The superscalar processor was simulated using a trace-driven timing model, similar to the one described by Wall et. al [6]. The timing model processes the program trace, looking at one instruction at a time. For each instruction, it assigns a timestamp for its progress through each stage of the pipeline. It keeps side structures to track occupancy of resources, etc. The timing model has been validated against a more detailed full-blown cycle accurate pipeline simulator, and the timing model is a very reasonable but much faster approximation. Figure 4.1 plots the reported performance in terms of instructions-per cycle (IPC) for a subset of SPEC benchmarks.

The timing model can be used to extract the information required to construct the program dependence graph. As described by Fields [11], critical path and slack analysis require a backward traversal of this graph. Rather than buffering up the graph for the full program run, the approach in this study is to periodically buffer a fragment of the dependence graph and perform the analysis on each fragment in isolation. This bounds both space and time requirements of the computation. The results of this approach were compared to those when the complete graph was analyzed in one go, and it was found that for buffer sizes of a few tens of thousands of instructions, the error induced by this approach is minimal. Also note that for critical path computation, a more exact analysis with low buffering requirements exists, which relies on the existence of “convergence” edges through which the critical path is guaranteed to flow [11]. However, the approach doesn’t extend to

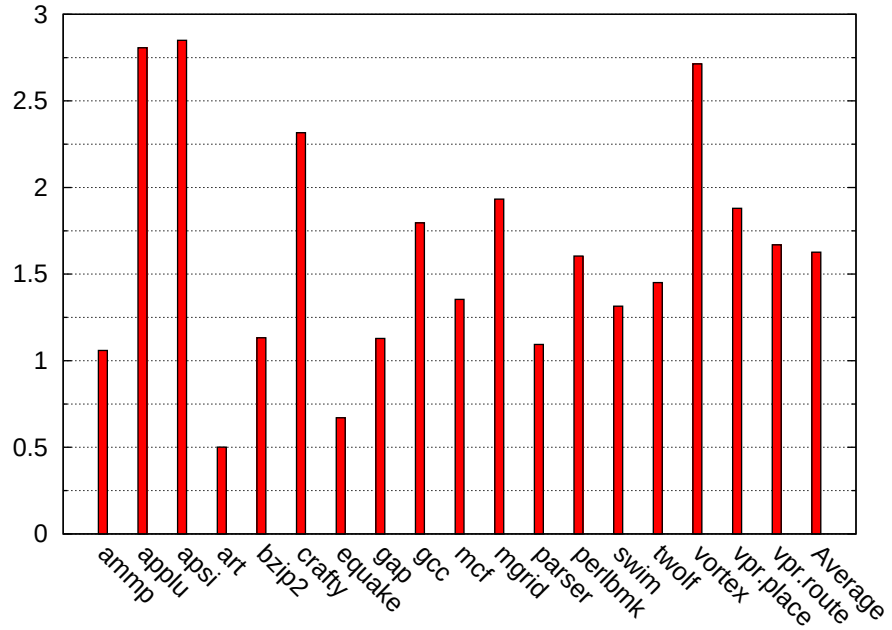

Figure 4.1: IPC achieved on SPEC benchmarks for simulated superscalar processor.

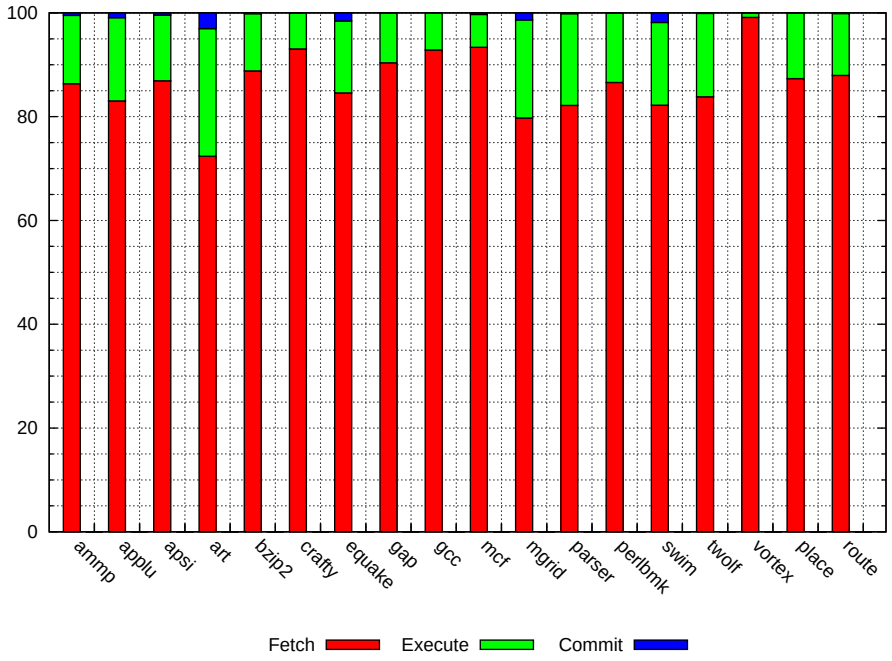

Figure 4.2: A breakdown of the nodes that comprise critical path.

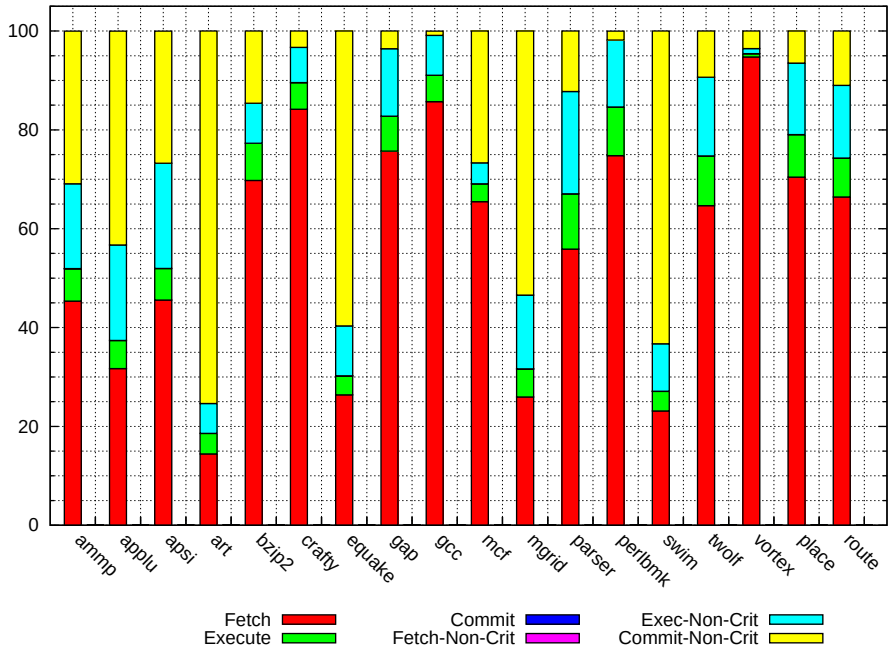

Figure 4.3: A breakdown of the instructions by their criticality behavior.

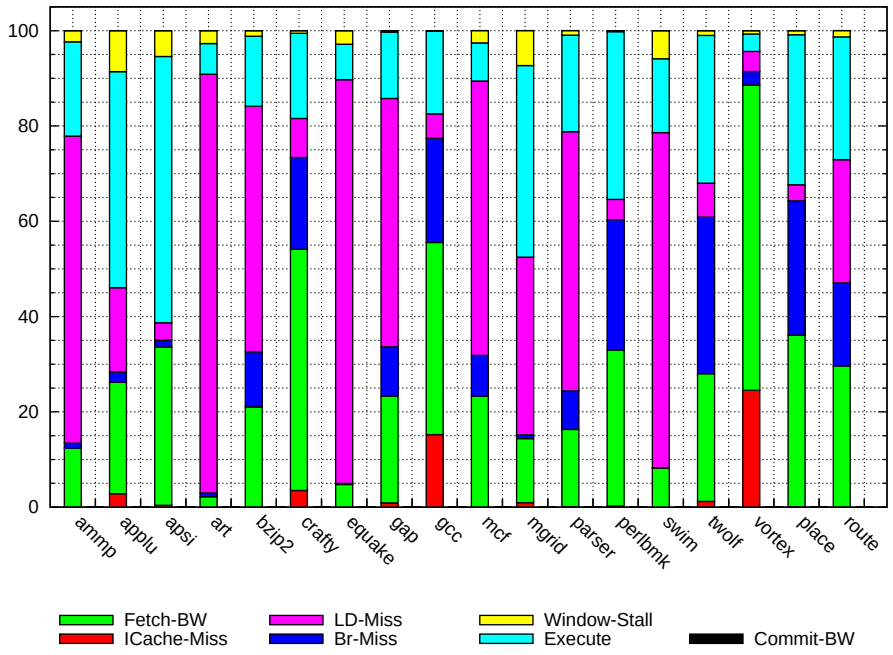

Figure 4.4: A breakdown of the critical path latency by different type of edges.

other analyses such as slack computation. Therefore that approach was not adopted.
Once the program critical path is found using the above (or any other methodology), there are multiple ways of summarizing the result to get useful insights. Figures 4.2, 4.3, and 4.4 show three ways of visualizing the critical path. Figure 4.2 shows the breakdown of nodes on the critical into the three possible categories: Fetch(F), Execute(E) and Commit(C). The figure shows that the critical path is consists mainly of critical fetch nodes. Several instructions can contribute multiple nodes to the critical path, and also several edges can jump over multiple instructions, that are non-critical. Figure 4.3 shows the breakdown of instructions in terms of their criticality behavior. This graph shows, for example, that most of the instructions on the *swim* benchmark are commit non-critical, implying that it suffers from a large number of reorder buffer stalls. In general, most of the critical instructions are fetch-critical.
Finally, figure 4.4 illustrates another way of characterizing the critical path, in terms of edges that comprise the critical path rather than the nodes. The edge latencies on the critical path are divided up into buckets depending upon the type of dynamic event upon which the edge latency could be blamed. For example, the latency on critical FF edges is listed under the *Fetch-BW* category, because it represents delay incurred due to limited fetch bandwidth. However, there is a class of FF latency that is listed separately corresponding to the F nodes that incurred an instruction cache miss, and that latency goes in the *ICache-Miss* category. Execute (EE and EC) latencies are reported in the *Execute* category, except for the load instructions that missed in the cache. The EE latency due to critical missing loads goes in the *LD-Miss* category. For branches that mispredict, execution latency on the EF edge going to the target (which might potentially include a mispredict penalty) is reported under the *Br-Miss* category. Long-latency load misses, long dependence chains, etc. can cause a buffer stall, and once space frees up in the buffer, there is a penalty to warm up the pipeline. This penalty is incurred by the critical CF edge and is classified as *Window-Stall* penalty. Finally, the latency on critical CC edges is placed under the *Commit-BW* class.

4.1.2 Prevalence of Fetch Criticality

Figure 4.3 shows that on an average across the SPEC 2000 benchmarks, about 57% of all instructions are fetch critical. About 19% of instructions are either execute-critical or have a critical EE edge jumping over them. Most of the remaining instructions in that configuration are commit non-critical, because of a window stall CF dependence jumping over them, and only a small percentage is commit-critical. Previous work [3, 21] has explored techniques to reduce the performance impact of execute-critical instructions. The underlying theme is to speed up execution of execute-critical instructions by giving priority in resource allocation decisions to likely critical instructions over likely non-critical instructions. This is based on the per-PC locality behavior of execute-criticality which can be exploited in dynamic mechanisms.
The above techniques which target execute-critical instructions tend to make a limited impact on performance. This is because much of the contribution (in terms of instructions) to the program critical path comes from instructions whose fetch node is on the critical path. This means that

attacking fetch-critical instructions could lead to large improvements in performance.
Fetch-criticality, however, is not as well understood as execute-criticality. This work tries to understand the reasons for large amounts of fetch criticality in superscalar execution, and makes the connection between fetch-criticality and existence of parallelism.
At its core, fetch-criticality arises because of the limitations imposed by superscalar architecture on program execution. In the absence of architectural constraints (dependences), a program could execute as soon as permitted by the longest chain of the enforced data- and control-dependences. In such a scenario, the critical path would consist of EE data-dependences, and EF control-dependences from mispredicted branches to their control-dependent (mispredicted) targets. Thus, the critical path would comprise mainly E nodes and some F nodes depending upon the branch prediction rate and the prevalence of control flow.
However, superscalar architectures don't allow independent instructions to be fetched and executed in any arbitrary order. In particular, the major constraints are the limitations of in-order fetching of instructions, limited buffer sizes within which instructions can execute out-of-order, and limited fetch bandwidth. These constraints lead to large amounts of fetch-criticality.

4.1.3 Fetch Criticality Generating Events (FCGEs)

There are three kinds of events that cause fetch-criticality in superscalar execution. These events will henceforth be referred to as *fetch criticality generating events (FCGEs)*. Fetch criticality generating events fall, roughly, into three categories, listed below and illustrated in figure 4.8:

- Fetch FCGE: The source of fetch criticality comes from reasons related to fetch of instructions. This includes instruction cache misses, which delay the fetch of an instruction. In addition, limited fetch bandwidth is also an FCGE, because it adds delay to the fetch time of future instructions.
- Execute FCGE: The source of fetch criticality comes from execution of previous instructions. This includes branch mispredicts, which delay the fetch of the correct target of the branch instruction. Similar behavior comes from other mispredictions such as for misspeculated loads.
- Commit FCGE: The source is reorder buffer stall, typically due to long-latency instructions or long dependence chains that cause the buffer to fill, stalling the fetch unit.

Note that all of the above FCGEs can cause large amounts of fetch-criticality because of the restriction of in-order fetch imposed by superscalar architectures. Therefore, once the fetch stream is delayed/stalled by an FCGE, the fetch of all future instructions is delayed, making it quite likely that the critical path flows through later fetch nodes as well. So, for example, mispredicted branches delay fetch of all future instructions, even future instructions that are control-independent of the branch. Thus even instructions control-independent of the branch can become fetch-critical even though they could have been fetched much earlier if instructions were not required to be

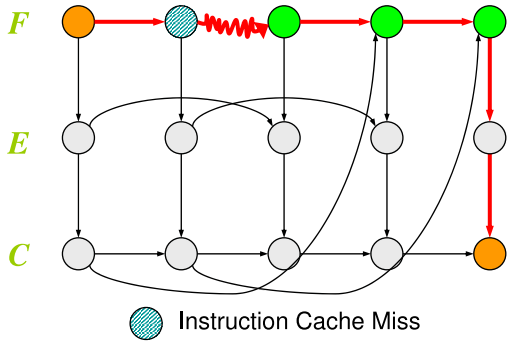

Figure 4.5: Fetch FCGE: Instruction cache misses extend the length of the path through the fetch node of one instruction, making subsequent instructions fetch-critical.

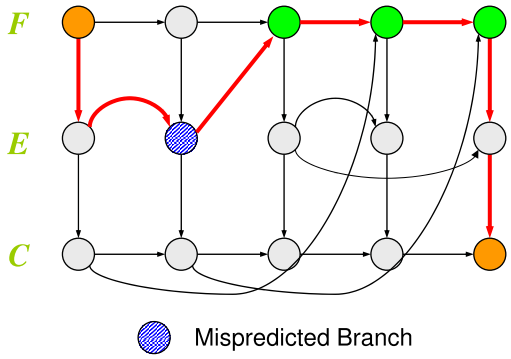

Figure 4.6: Execute FCGE: The correct target instruction of a mispredicted branch cannot be fetched until the branch instruction is executed to detect the incorrect prediction.

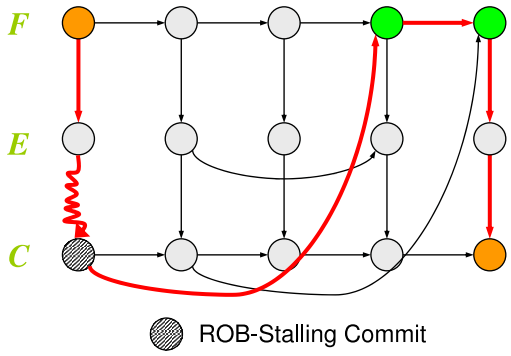

Figure 4.7: Commit FCGE: When the reorder buffer or scheduler is filled to capacity the fetch unit must be stalled. This can make the first stalled instruction fetch-critical.

Figure 4.8: Examples of the three type of FCGEs.

fetched in-order. Similarly, instruction cache misses delay fetch of all future instructions, even if the future instructions are present in the instruction cache.

4.2 Fetch Criticality ⇒ Unexploited Parallelism

As explained in chapter 2, parallelism can be created by removing some of the previously sequentializing constraints. Parallelization techniques can relax/remove dependences on the critical path and therefore improve performance in such cases. As far as an architect is concerned, the objective is to have the critical path comprise mainly of the application dependences and minimize the impact of architecture-imposed dependences. Parallelization techniques can remove architectural dependences (besides sometimes removing application dependences as well) to move the critical path closer to that goal.

The existence of fetch-criticality implies the potential for parallelism by removing in-order fetch dependences. Large amounts of fetch-criticality indicate that the achieved dependence height is being dictated by architecture-imposed in-order fetch dependences rather than just application control- and data-dependences. Several parallelization techniques are possible to address the impact of the three FCGEs. In superscalar design, architects have used techniques like higher fetch-bandwidth, larger instruction caches, better branch predictors and larger reorder buffers to reduce fetch-criticality by extracting larger amounts of "Instruction Level Parallelism" (ILP). In addition, in-order fetch dependences can be alleviated by out-of-order fetch architectures, such as control-independence architectures [27–30], speculative parallelization architectures [31–40], or through explicit parallelization of applications for execution on multicore or multithreaded architectures.

Note that parallelism can also exploited by some techniques that address execute- and commit-criticality. Some parallelization techniques can reduce execute-criticality to create parallelism, such as by reducing the number of control- and data-dependences enforced by the architecture. For example value prediction techniques can break true data-dependences in some cases. Better branch prediction can reduce the enforced control-dependences. Register and memory renaming techniques can also remove execute dependences. All of these techniques create parallelism by attacking execute-criticality. These techniques have been quite successful in parallelizing execution in the past. However, new advances are coming at a slow pace in these directions, partly because extending current techniques requires devoting larger amounts of area to branch/value prediction, etc. and more complex circuits for memory renaming. Designers typically deem the additional returns not worth the investment in chip area and the power/complexity costs incurred.

In addition, commit-criticality could also potentially become a problem, even though it is not a major limiter for superscalar processors. Parallelism in those cases can be created through higher commit-bandwidth, or by removing some of the in-order commit edges. But in practice, processors make the peak commit bandwidth match the peak fetch-bandwidth of the machine, and

commit-bandwidth is rarely a problem. In-order commit can sometimes restrict parallelism, but is useful to provide sequential semantics to the external world.

4.3 Task-Based Parallelization to Alleviate Fetch-Criticality

Section 4.2 identifies several approaches to exploit parallelism by alleviating fetch-criticality. These fall in two classes: a) extract more ILP in superscalar processors and b) explore out-of-order fetch techniques. The first approach involves techniques like increasing the fetch width, larger scheduler and reorder buffer sizes, larger branch predictors, etc. Typically, these changes go together to keep the architecture balanced otherwise one FCGE can dominate and hide the benefit of a technique that addresses another FCGE. However, these techniques are not very attractive because current superscalar processors are at a point where investing chip area into these techniques leads to very low returns on investment, if any at all. These techniques can also lead to increases in power consumption and design complexity.
The second approach involves out-of-order fetch and is more promising. It can lead to scalable architectures that yield good return on chip area investments. One of the ways to achieve high performance in this domain is to divide up the program into multiple “tasks” or “threads” that can fetch and execute concurrently on different cores/threads of a multicore or multithreaded architecture. Such an approach can reduce fetch-criticality by exploit parallelism through concurrent tasks. The primary focus of this thesis is on implicit tasking systems that externally maintain sequential semantics, but internally (and dynamically) partition the application into tasks for simultaneous fetch and execution on a speculative parallelization architecture. However, several of the insights developed here could potentially be applied to other flavors of task-based parallelization.

4.3.1 Control-Independent Task Spawning

This thesis focusses on tasks that are control-independent of instructions that spawn them. This section explains the concept of control independence and terminology associated with spawning a control-independent task.
In a program flow graph, an instruction X is said to *postdominate* another instruction A iff all paths through the flow graph from A to the exit pass through X [41]. In other words, X postdominates A if X is guaranteed to execute after A executes, regardless of intervening control decisions. As an example, in the flow graph of Figure 4.9, block E postdominates block A. The block E is therefore *control-independent* of the branch in block A. A binary rewriter can efficiently calculate all the postdominators of all branches in an executable of size $O(n)$ in time $O(n \log n)$ [42].
Architectures can exploit control-independence property of branches to spawn future control-independent regions of the program as tasks that can execute concurrently with the main stream of execution. Thread-level Speculative and Speculative Multi-threaded processors find parallelism in a single thread of execution by breaking it into multiple tasks that are executed

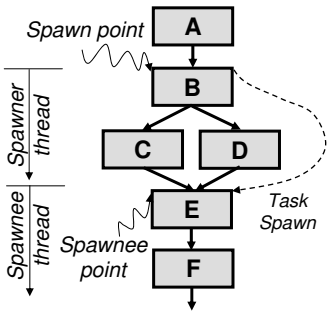

Figure 4.9: Terminology to describe a control-independent task spawn.

concurrently [31–35, 37–39, 43]. The control-independence property can also drive the choices of spawns in these machines and has been shown to be subsume heuristics like loops, procedure calls, and their continuations [36].

For example, in Figure 4.9 a new task could be created at block E whenever block B is reached, since the processor is guaranteed to reach block E at some time in the future. In this case, the new task generated starting at block E would be called the *spawnee* task with E being the spawnee block, and B the *spawner* block. The spawner and spawnee tasks can concurrently fetch instructions. When the spawner task reaches block E, it stops fetching instructions (the work it is about to begin has already been done by the spawnee task). At this point, the spawner task *reconnects* with the spawnee.

Instructions in the spawnee task that depend on data produced in spawner task can be handled in a variety of ways. They can be speculatively executed assuming that the data is available[34, 35], and signal a misspeculation if a violation is later detected. Or a data-dependence predictor [44] can be used to identify such instructions, which can then be delayed (synchronized), until data value is (conservatively) released by the spawner task. The next section presents a dependence graph model that captures the constraints that arise in spawning a control-independent task.

4.3.2 Dependence Graph Model for Control-Independent Task Spawn

This section describes how Fields' dependence graph model for superscalar execution can be extended for a speculative parallelization architecture that does control-independent task spawns. This section refers to such architectures as Control-Independence architectures.

Control-independence architectures remove the restriction of in-order fetch at task boundaries that were previously imposed by superscalar architectures. On the other hand, delay might be added to inter-task dataflow. Exactly how much delay is added depends upon the particular data-dependence handling technique used. Further, techniques such as data speculation can lead to task squashes when speculation fails.

Table 4.2 lists the edges that need to be added to the superscalar dependence graph model of Fields et al, in order to model the effects of spawning a task. The first modification is a new FF edge going from the fetch node of the *spawner* to the fetch node of the *spawnee*. Note that there is no longer a

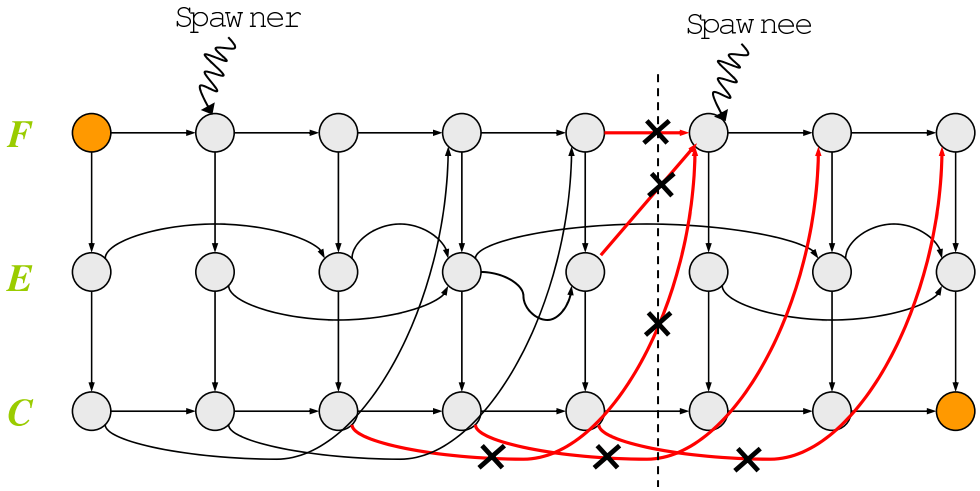

Figure 4.10: **Before Spawning**. Edges marked with an "X" will be removed. These include the FF edge from last instruction in spawner task to first instruction in spawned task, EF edge if the last instruction in the spawner is a mispredicted branch, and rob_size CF edges crossing the task boundary.

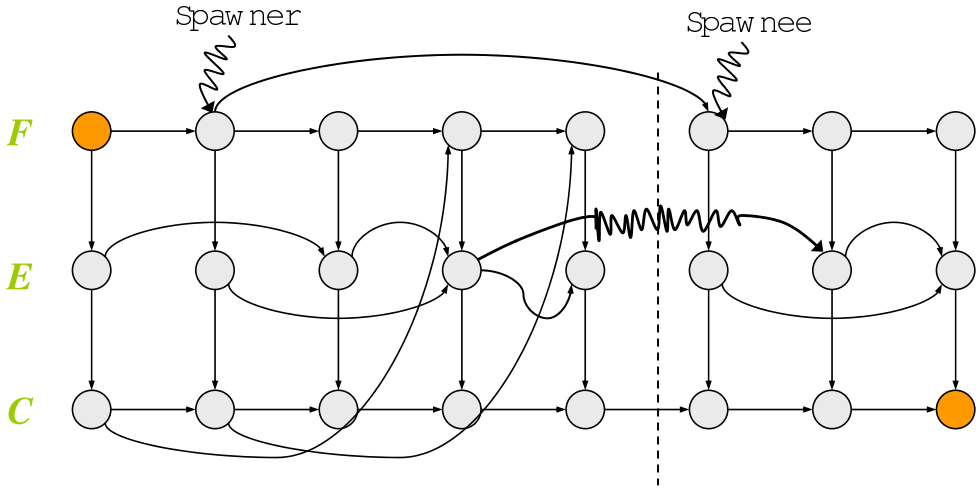

Figure 4.11: **After Spawning**. A spawn FF edge is added. Additional latency might be added to EE edges that cross task boundary.

Figure 4.12: Impact of control-independent task spawning on the program dependence graph.

fetch edge from the last instruction in a spawner task to the first instruction in the next task, which is another modification from the superscalar model. This means that the spawnee can start fetching as soon as the spawner fetches (after some penalty, equal to the latency on the particular FF edge, which can be used to model a *spawn penalty*). Intra-task fetch proceeds as before.

The second modification is that if an EF edge crosses the task boundary, it is removed. The EF edge would exist if the last instruction in the spawner task was a mispredicted branch and the first

Name	Constraint modeled	Edge	Comment
FF	In-order fetch **(non-spawn)**	$F_{i-1} \rightarrow F_i$	Non-spawnee instruction i cannot fetch before $i-1$.
FF	**In-order fetch (spawn)**	$F_{i-s} \rightarrow F_i$	Spawnee instr i cannot fetch before spawner $i-s$ (s = spawn distance).
CF	Finite reorder buffer size	$C_{i-r} \rightarrow F_i$	**Instr $i-r$ and instr i are in the same task** and r is the size of the reorder buffer
CF	**Finite task resources**	$C_{last(i-n)} \rightarrow F_{first(i)}$	Instr $last(i-n)$ is the last instr in task i-n, and instr $first(i)$ is the first in task i and n is the number of task contexts
EF	Failed Speculation	$E_{i-1} \rightarrow F_i$	Instr $i-1$ is a misspeculating instruction (mispredicting branch/load, etc), **and i is in same task**.
DE	Execution follows fetch	$F_i \rightarrow E_i$	An instr cannot execute before it has fetched.
EE	Data dependences	$E_j \rightarrow E_i$	Instr j produces an operand of i. **Inter-task data flow can have longer latency than intra-task data flow.**
EC	Retire follows execution	$E_i \rightarrow C_i$	An instr cannot retire before execution.
CC	In-order retirement	$C_{i-1} \rightarrow C_i$	Instr i cannot retire before $i-1$.

Table 4.2: FRB dependence rules [3] adjusted for control-independent task spawning (**bold**).

spawnee instruction was the correct target. Since the spawnee task was control-independent of the spawner instruction and was fetched out-of-order, the mispredicted branch no longer delays fetch of the spawnee. Therefore, the EF edge is removed.

Third, CF edges impact fetch only within a task, and not across tasks. This means that back end stalls in one task need not stall fetch in the successor tasks, effectively allowing for a distributed window of instructions. In addition there is another CF edge to model finite task resources.

Fourth, the Fields' model already contains an EF edge to model branch mispredicts. We generalize the notion of mispredicts to capture intra-task store-load violations, as well as inter-task violations due to failed data speculation. Frequent data misspeculation can also be a FCGE, and can be treated in a similar manner as branch mispredicts. Note that the latency for detecting and communicating the failed speculation is implicitly captured through the weight given to this edge.

Finally, depending on the specific policy for handling inter-task data dependences for a given control-independence architecture, delay might be added along EE edges that cross task boundaries. This can model, for instance, the latency of inter-core communication, as well as the delay for architectures that synchronize on data dependences.

4.4 Necessary Conditions for Existence of Parallelism

4.4.1 Rules for Successful Task Spawn

A task spawn is said to be successful if it improves performance over superscalar execution. The conditions for a successful task spawn can be summarized by two simple rules:

1. The first (in program order) critical instruction in the spawned task is fetch-critical.
2. The slack on all edges that cross the task boundary after (and as a result of) the task spawn must be non-zero. Further, the slack on the EE edges that cross the task boundary must be greater than the data latency added by the spawn mechanism.

As described in Section 4.3, control-independent spawning can be profitable by alleviating fetch-criticality in applications. Rule 1 states that if the spawnee point is not fetch-critical in the first place, then the spawn is largely useless since it tries to address a problem that does not exist (it could, however, create new problems by delaying EE edges that don't have enough slack).
Rule 2 states that it is not enough to break fetch-criticality chains, the objective of spawning should be to speed up program execution. This means that the new program critical path should not be worse than the original path (that passed through fetch node of the spawnee). In particular, the only reason why it could be worse off than the original path, could be if it flows through one of the EE edges that are delayed due to the act of spawning.

4.4.2 Spawn Rules in Action

This section illustrates the above rules through a detailed example from the SPECInt2000 benchmark *twolf*. Figure 4.13 shows the control-flow graph of a fragment of interest in the function *new_dbox_a*, which accounts for a large fraction of the overall execution time. The node marked A is a branch that is likely to mispredict and generate fetch-criticality. We find that its postdominators: C, D, E, F, and G, are all likely to be fetch-critical (note that blocks D and F also contain FCGEs). In general we find that postdominators of blocks containing low-confidence branches have a high likelihood of being fetch-critical. Thus a dynamic instance of any of of these postdominators, when spawned from A, is likely to satisfy Rule 1 for spawn success.
On the other hand, not all of these postdominators are likely to satisfy Rule 2. Figure 4.13 also shows dataflow edges that are likely to be on the program critical path. In particular, the statement that produces *rowsptr* in block C is on the backward slice of likely-to-mispredict branches in blocks D and F. Thus, these EE edges from C to D and C to F are likely to have very little slack, since almost all mispredicted branches are on the program critical path in a superscalar processor. Tasks which add delay to these edges are likely to violate Rule 2.
In this case, we find that when we spawn D, E, or F from A, one (or both) of the EE edges flowing the value of *rowsptr* out of C gets delayed. Therefore, we can rule out these tasks as unprofitable options. On the other hand, if we spawn C or G, we find that the EE edges that cross the task

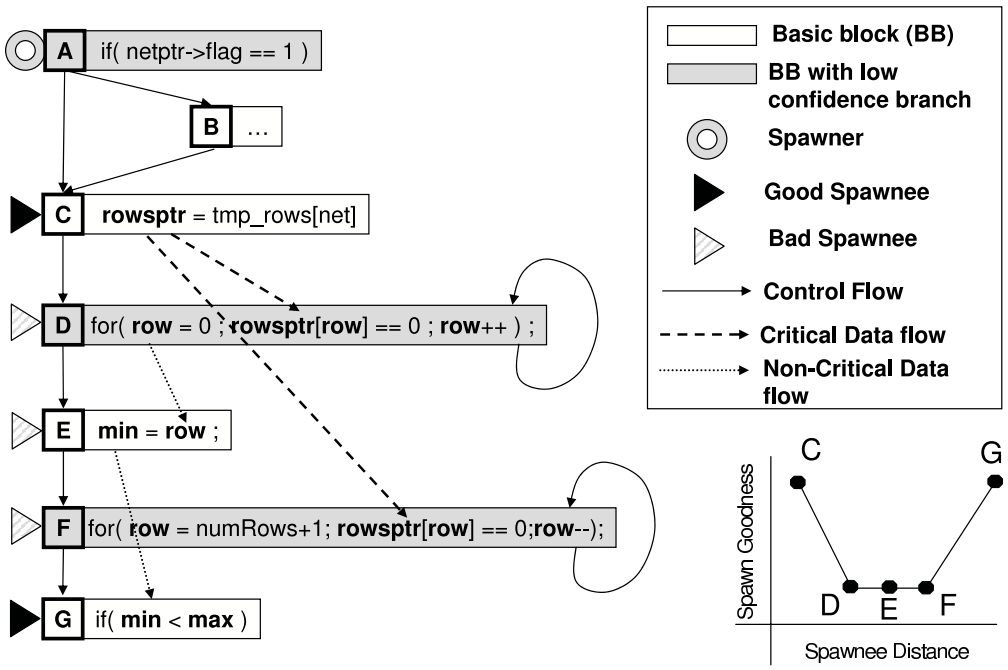

Figure 4.13: An interesting example from the `new_dbox_a` subroutine in twolf (somewhat simplified), showing spawnee choices for low confidence branch A. Spawnees D, E and F introduce inter-task data-dependence edges (through rowsptr) leading into already execute-critical nodes, and are bad. C and G are profitable tasks, since the subsequent execute-critical instructions are data-independent.

boundaries have a lot of slack. In particular, when we spawn G, the edge corresponding to the value *min* that flows from E to G has a lot of slack since the branch in G is usually predicted correctly. So even though the $E \rightarrow G$ EE edge is delayed, the delayed execute node of G is unlikely to be on the program critical path.

This is further explained through a set of criticality diagrams. Figure 4.14 shows the critical path in a normal superscalar execution of the program. Branch mispredicts of A, D, and F are on the program critical path. Control-independent spawning is an attractive proposition to reduce the impact of misprediction of A on fetch-criticality of its control-independent instructions. In order to achieve this, we can spawn one of the CI points of A, which are likely to be fetch-critical (thus satisfying Rule 1). The available spawnee choices are: C, D, E, F and G. Note that for simplify, the diagrams don't show several dataflow edges that contain a lot of slack because they don't affect the profitability of these tasks.

Figure 4.17 shows the outcome of spawning C from A. This is a profitable spawn that succeeds in reducing criticality. The instruction C was originally fetch-critical because of the mispredicted branch A. Spawning removed the execution of mispredicted branch A as well as fetch of instructions between A and C from the critical path.

On the other hand, spawning D from A, shown in figure 4.15, is an unprofitable task that violates

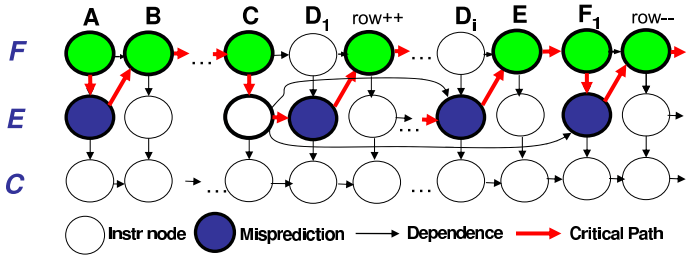

Figure 4.14: **Superscalar execution:** Branch mispredicts cause lot of fetch-criticality.

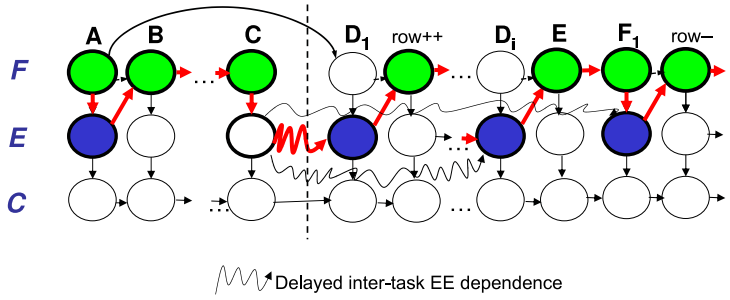

Figure 4.15: **Failed Task (Rule 1)**: Instruction D was not fetch-critical in superscalar. Critical path is worse due to delayed inter-task dataflow for *rowsptr*.

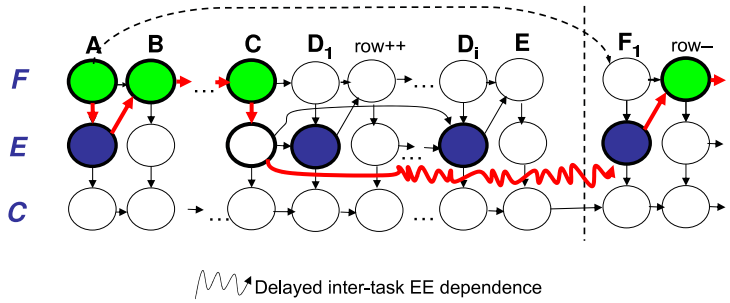

Figure 4.16: **Failed Task (Rule 2)**: More delay is added to a near-critical EE edge than slack on it.

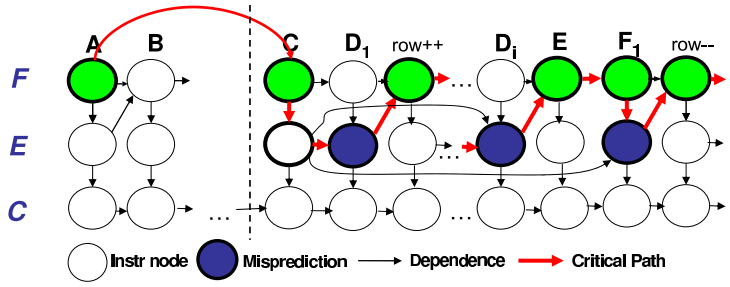

Figure 4.17: **Successful Task**: Spawning C from A shortens the critical path by fetches C faster than earlier. It also removes execution of A (mispredict) and fetch of B from critical path.

Figure 4.18: Critical path for superscalar execution and different spawn scenarios. *CF* edges don't contribute to critical path and are not shown for ease of understanding.

Rule 1 for spawn success. D was not fetch-critical in superscalar execution, so fetching it faster didn't help. But spawning off D adds a large latency to the already critical EE edge from C to D, delaying dataflow into execution of the branch mispredict D. Thus we delayed an already critical EE edge due to a spawn, making performance worse. Spawning F from A is another unprofitable task and is illustrated in figure 4.16. Even though it satisfies rule 1, it fails to satisfy rule 2. This is because a large amount of delay is added to the inter-task EE edge from C to F, which is more than the slack on that near-critical edge.

4.4.3 Proof of Spawn Rules

This section gives a formal proof of the rules for successful task spawning.
Claim: A task spawn can lead to improvement in performance (by decreasing the length[1] of the program critical path [3]) **only if** the following conditions hold:

1. The first (in program order) critical instruction in the spawned task is a fetch-critical instruction.

2. The slack on all edges that cross the task boundary after (and as a result of) the task spawn must be non-zero. Further, the slack on the EE edges that cross the task boundary must be greater than the data latency added by the spawn mechanism.

Assumptions: This proof assumes that a task spawn can only introduce the following changes to the program dependence graph:

1. Spawning breaks the in-order fetch edge at the task boundary (from F node of last instruction in spawner task to F node of first instruction in spawned task). If the last instruction in spawner task was a mispredict, spawning also breaks the edge going from E node of that instruction to the F node of first instruction in spawned task.

2. Spawning breaks rob_size CF edges going from C node of an instruction in the spawner task to F node of an instruction rob_size later in the spawned task. The original edges represented resource dependences due to finite ROB when these instructions were fetched in the same task.

3. Spawning can add extra latency to EE edges that cross spawn boundary. These represent delayed inter-task data dependences, with the amount of delay varying depending upon the specific dependence handling techniques used.

4. A spawn FF edge is added from the fetch (F) node of the spawner instruction to the F node of the first instruction in the spawned task. The latency on the edge represents a startup cost.

[1]The length of a path in the program dependence graph is defined to be sum of latencies on all of its edges.

Figure 4.12 illustrates these modifications. It is assumed that all of the other edges in the program dependence graph as well as their latencies stay unchanged. The validity and impact of these assumptions are examined later.

Proof: This proof proceeds by showing that if either of the two conditions is not met, then spawning the task can not lead to a reduction in the critical path length.

Case 1: Suppose the first (in program order) critical instruction in the spawned task is not fetch-critical. This means that it is either execute-critical or commit-critical, with the last-arriving dependence edge incoming from an instruction node before the start of spawned task.

This dependence edge cannot be one of the edges removed by the task spawn, since all such edges end in the F node of some instruction in the spawned task, and this edge ends in a non-F node. Thus, the latency of this edge stays unchanged or increases as a result of the spawn. Further, the latencies on all of the other edges in the original program critical path stay unchanged. Hence, the sum of the latencies on the original program critical path remains the same or increases. This implies that there is at least one path in the new dependence graph after spawning whose length is greater than or equal to that of the original program critical path. Thus, the length of the program critical path is not decreased due to this spawn.

Case 2: Suppose the latency added on at least one of the EE edges that cross the boundary as a result of the task spawn is greater than or equal to the slack on that edge. We will now show that the program critical path definitely got longer due to this spawn.

Let us pick one of those edges, say e, such that it originally had a slack s. Suppose a delay l was added to e such that: $l \geq s$. From the definition of slack, we know that in the original program dependence graph, there was at least one path P that included e, such that increasing the latency of e by s made P longer than the original critical path.

Now, when we do that spawn, the latencies on all the edges in P stay unchanged, except for the latency of e. Note that P couldn't have included any of the CF or FF edges that were removed by the spawn, or any of the other EE edges that crossed task boundary created by this spawn. This is because there can be only one edge on a path that crosses the boundary created by the task spawn. For P, this edge is e. And since the latency added on e, $l > s$, P is now longer than the original program critical path. Thus, the length of the program critical path is not decreased due to this spawn.

Conclusion: Combining the two cases, we have shown that if either of the above conditions don't hold true, the spawn cannot lead to a reduction in the critical path length. Hence a spawn can be profitable only if the above two conditions hold true. Note that the converse of this result doesn't necessarily hold. That is, even if the above two conditions are true, a spawn may still not be profitable. This might happen because a near critical path may be of the same length as the program critical path, or the delay added to one of the inter-task EE edges might be exactly equal to its slack (minus 1 cycle to be precise). In that case, doing the spawn will keep the critical path unchanged.

Impact of Assumptions: Note that in a real system, the assumptions made above about the impact of spawning on the program dependence graph don't always hold true. Spawning a task on a

different core can suffer warm-up effect which can change the cache and branch predictor behavior. The spawned task can suffer additional cache misses and branch mispredicts since the local caches and branch predictor on the core might need to be warmed up. Spawning the task can also create additional pressure on the memory subsystem by posing higher bandwidth requirements which can lead to higher latency on dependence edges that involve memory operations. On the other hand, the new task has a separate L1 cache available to it, so capacity misses might be decreased. In addition, spawned task might suffer data misspeculations. The net result is that the spawned task is squashed, and a misspeculation (EF) edge is added to the dependence graph. Latencies on some of the other edges might also change slightly, especially those that are impacted by contention for resources such as issue slots since the contention behavior will be changed. While the above factors are not incorporated into the model, chapter 5 demonstrates that their effects are amortized over the long-run. Therefore, they can be ignored without having too much of an impact on predictions about task behavior and parallelism. Further, ignoring these second-order factors simplifies the treatment of parallelism and policies based on these rules perform well in our system. So, to a first order approximation, the assumptions made here are reasonable to understanding the system.

CHAPTER 5

QUANTIFYING PARALLELISM FROM POTENTIAL TASKS

Chapter 4 presented a dependence graph model for control-independent task spawn, as well as the relationship between fetch-criticality and parallelism. This chapter develops those insights into a quantitative model of task parallelism. The objective is to quantify the expected benefit from a potential task choice. The benefit might come from one of the several “sources” of parallelism. Further, there are costs to spawning a task that might reduce or even completely swamp the benefit from parallelism. The model developed in this chapter accounts for the important factors to make its predictions.

5.1 Task Benefit and Critical Path Length

The act of spawning a new task causes the program dependence graph to change as described in section 4.3.2. As a result of these changes, the length of the program critical path after spawning changes from what it would have been in absence of spawning (superscalar execution). The performance improvement due to a particular task spawn is then simply the decrease in the critical path length because of that spawn.

One approach to quantifying this decrease is to measure the length of the critical path for two cases: one for superscalar execution, and another one where the task of interest is spawned. Subtracting the superscalar critical path length from the length for task spawn case would provide the benefit of spawning the task. However, this is a very computationally expensive approach since there are an extremely large number of tasks options available in most applications, and for each option, there can be several dynamic instances within a given program region.

An alternative approach is to have an estimation model that trains itself on the original superscalar execution and can make prediction about the expected benefit of spawning a task. This chapter develops such a model based on the insights of the previous chapter. The model can accurately estimate the task benefit “in-place” without actually spawning the task.

5.2 Assumptions About Impact of Tasks

The model makes some assumptions about how spawning a task impacts the dependence graph of superscalar execution. In particular, it assumes that spawning a task causes only the following changes to the original superscalar graph (illustrated in figure 5.1):

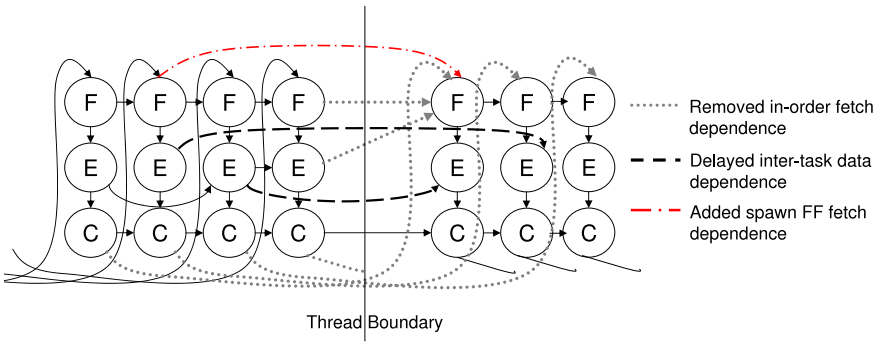

Figure 5.1: Modifications to the Dependence Graph caused by spawning a task.

- In-order fetch dependence edges (F → F, E → F and C → F) are removed at the task boundary,
- An F → F spawn edge capturing out-of-order fetch is added from the spawner to the spawned instruction, and
- Delay is added to EE edges at the boundary due to synchronization and communication required for inter-task data dependences.

The rest of the dependence graph edges and latencies are assumed to stay exactly the same. Section 4.4.3 described some reasons why these assumptions may not always hold in a real system. This chapter will show that the assumptions still lead to a good approximation of the system's behavior, especially for tasks that are spawned often.

5.3 Estimating Task Benefit

5.3.1 Definition: Adjusted Slack

In order to estimate the benefit from a task, the term *adjusted slack* on an edge will need to be defined. Recall that on a dependence graph of program execution for a given architecture, the *slack* on an edge is the difference between the length of the program critical path, and the longest path including that edge.

As described in section 5.2, spawning a task removes some dependence edges at the task boundary but for the remaining edges crossing the task boundary, some delay might be added due to communication and/or synchronization penalties. The *adjusted slack* on any such edge is the slack on the edge in the original superscalar graph minus the delay added to that edge in the resulting graph from spawning the task. This gives the difference between the length of the original critical path of superscalar execution, and the longest path in the new graph that includes this edge after the task spawn (this follows from the assumptions of section 5.2).

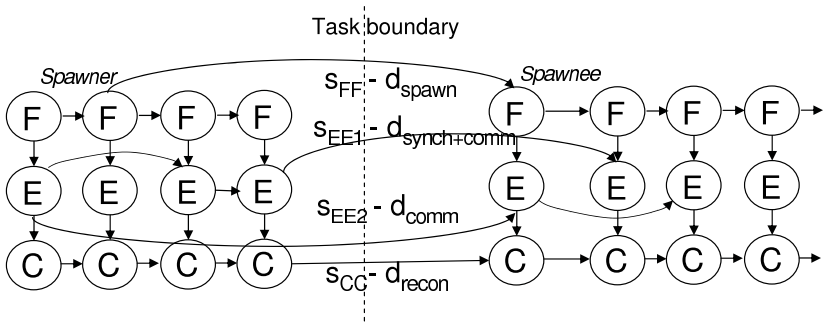

Figure 5.2: The benefit from a task is bounded by the slack on edges crossing task boundary. The spawn F → F edge suffers a spawn penalty, the C → C edge suffers a reconnection penalty. EE edges suffer a communication penalty, and potentially a synchronization penalty. C → F edges don't cross task boundary and are not shown.

5.3.2 Performance Benefit from Spawning a Task

As described before, the benefit from spawning a task is the reduction in program critical path length from doing the spawn. Spawning a task removes some edges from the original graph at the task boundary. After spawning a task, the new critical path can cross task boundary in only three ways, corresponding to the three ways in which the dependence graph changes:

1. The new critical path can flow through an E → E data-dependence edge crossing the task boundary. The task benefit is the difference between the original critical path length, and the length of the longest path in the new graph that *includes* the E → E edge. This is simply the *adjusted slack* on the edge, adjusted for communication/synchronization delays. Section 5.3.3 gives an example of calculating adjusted slack for E → E edges.

2. It can pass through the commit C → C edge at the task boundary. This is similar to the first case, and the benefit is the *adjusted slack* on the C → C edge, adjusted for the delay of passing the commit token between tasks, which is typically a fixed quantity.

3. It can flow through the spawn F → F edge from the spawner to the spawnee instruction. This is a trickier case, since the spawn F → F edge didn't exist in the superscalar graph. However, it is still possible to calculate the *adjusted slack* on the edge if it had been present in the original graph. Section 5.3.4 describes how to compute this quantity.

Figure 5.2 summarizes the situation. The improvement in the program critical path is constrained by the edges that cross the task boundary after the spawn. A spawned task cannot decrease the length of the program critical path beyond the slack on any one such edge (adjusted for delays/penalties). The benefit of a task, then, is bounded by the minimum of the adjusted slack on all edges that cross the task boundary.

As a corollary, a spawned task cannot help performance if the original critical path crossed the task boundary through an E → E or the C → C edge, since the latency on these edges cannot decrease after spawning. That is, it can help only if the first critical instruction in the spawned task was fetch-critical.

5.3.3 Adjusted Slack Calculation for Synchronized E → E Edge

This section gives an example of how to adjust slack on E → E edges for synchronization delays. This is illustrated through a conservative synchronization policy: for data-dependences that cross task boundary, the value is released from a spawner task to the successor spawnee task when the following three conditions are met:

1. All branches in the spawner task have completed execution (thus data produced by bad-path instructions fetched beyond misspeculated branches is never released).
2. The spawner task has fetched and renamed (down the correct path) all instructions (in program order) prior to the first instruction in the spawnee task (ensures that the last writer has been seen).
3. The producer of the value has completed execution.

When the first two conditions are satisfied, the tasks are said to have reconnected. Suppose the time at which all branches in the spawner task have finished execution is t_{branch}, the time when all instructions in spawner task have been fetched is t_{all}, and the reconnection penalty is $penalty_{recon}$. Then the reconnection time is given by:

$$t_{recon} = max(t_{branch}, t_{all}) + penalty_{recon} \tag{5.1}$$

Suppose the original time at which the value was ready and the EE edge arrived at the consumer was t_{ee}, the reconnection time was t_{recon} and inter-task communication latency was t_{lat}. Then the delay added to the EE edge due to the synchronization policy is:

$$delay_{synch} = max(t_{ee} + t_{lat}, t_{recon}) - t_{ee} \tag{5.2}$$

For producer-consumer pairs that jump over multiple tasks, the value is released only after all tasks in between have reconnected. Therefore the delay calculation has to consider all intermediate reconnections. The adjusted slack on the EE edge then is:

$$s'_{ee} = s_{ee} - delay_{synch} \tag{5.3}$$

The above equations make it possible to compute the adjusted slack from the superscalar graph without requiring to build a new dependence graph or to recalculate any edge slacks. The adjusted slack computation requires tracking the execution times of branches and the fetch time of last

instruction in the spawner task. It should be possible to similarly incorporate other synchronization/data-dependence policies in this model.

5.3.4 Adjusted Slack for Spawn F → F Edge

This section shows how to estimate the adjusted slack on the spawn F → F edge from the original graph and without building a new graph. Let the timestamp of the spawnee F node *in the superscalar graph* be $t_{F\_spawnee}$ and that of the spawner be $t_{F\_spawner}$. Let the delay for a task spawn be $penalty_{spawn}$. Spawning the task causes the spawnee instruction to be fetched sooner by t_{gain}, which is:

$$t_{gain} = t_{F\_spawnee} - (t_{F\_spawner} + penalty_{spawn}) \tag{5.4}$$

This is because originally the spawnee was fetched at $t_{F\_spawnee}$. After being spawned, it could be fetched $penalty_{spawn}$ after the spawner was fetched. Let the original slack on the spawnee F node be $slack_{F\_spawnee}$. Then, the slack on the spawn F → F edge *in the superscalar graph* would have been:

$$slack_{FF\_spawn} = slack_{F\_spawnee} + t_{Fetch\_gain} \tag{5.5}$$

The spawnee F node already had a slack of $s_{F\_spawnee}$. Because of the spawn, it was fetched sooner, the two are combined to find the slack on the spawn F → F edge if it had existed in the original graph.

5.4 Overall Approach

This section summarizes the overall approach to estimate benefit of potential tasks as well as computational requirements for the model. Rather than analyzing the whole application trace, the model processes it in small segments to reduce memory footprint. The model constructs a dependence graph for the segment of the executed program to be analyzed. The dependence graph is built for superscalar execution. Slack information is then computed for dependence edges and nodes in one backward traversal of the dependence graph. Side structure track the execution time of branches and EE edges that can cross potential task boundaries.

The next step involves a forward traversal of the graph to estimate the performance benefit of all potential task spawn choices. For any task pair to be evaluated, the model computes the adjusted slack of edges crossing task boundary. The performance benefit of that spawn choice is estimated to be the minimum of the slack on all edges crossing task boundary. A structure tracks the aggregate information for each spawn (spawner, spawnee) pair evaluated. After all the segments have been analyzed, average statistics can be reported for each task. Later chapters show how to use this information to make a task selection, and how to limit the computational requirements of this approach by infrequently sampling program segments.

As far as computational requirements go, the model incurs a base cost and a cost per task. The base cost is incurred to build the dependence graph, and compute slack on each dependence edge. Computing the slack requires a backward traversal of the dependence graph, and takes time that is O(V+E), where V denotes the nodes in the graph, and E denotes the number of edges in the graph. Additional cost is incurred for each spawn pair whose benefit needs to be estimated. Computing the adjusted slack on spawn FF edge and commit CC edge requires O(1) work. Adjusted slack calculation for EE edges requires some more work. The release policy described in section 5.3.3 requires calculation of the reconnection time which in turn needs the execution times of all branches between the spawner and the spawnee. This requires O(V) time in the worst case. In practice, a separate sorted list of branches can be maintained for lower cost. And finally, slack information is needed for all the EE edges crossing the task boundary. This can be O(V) work in the worst case. To further reduce cost, we only need to look at the EE edges that have a small amount of slack, since the edge with the minimum slack is the one that matters. Sampling small segments in the program can further help keep the value of V small.

5.5 Validation

5.5.1 Infrastructure and Methodology

To validate the task performance model, this section shows simulation results for the Polyflow speculative parallelization system with 4 cores. The details of the system are described in chapter 6. The baseline is a superscalar processor with same resources as available to one core of the Polyflow system, except that it has the whole L2 cache available to itself. The specific parameters are given in table 5.1.
Validation is done by comparing the predictions from the model to measurements on an execution-driven simulator for the Polyflow architecture. The simulator uses a variant of 64-bit MIPS ISA that does *not* have any special instructions to support multi-threading. Task spawn points are obtained from a postdominance analysis performed on the program binary. Spawns points whose average length exceeds twice the modeled reorder buffer size are discarded because they would likely cause load imbalance. This section present results from SPEC2000 benchmarks that can be compiled on our toolchain. The simulations were done for representative intervals in the lgred [45] or train inputs whose function profile closely matches the overall profile for the ref input.

5.5.2 Validation Results

To assess accuracy of the task benefit model, this section compares predictions made by the model for individual task options to the observed performance improvements when that task is actually (and in isolation) spawned on the underlying system. For this experiment, the model analyzes program segments of 100K instructions at a time.
Figures 5.3 to 5.36 compare, for each task, predicted versus measured performance improvement.

Parameter	Value
Pipeline Width	4 instrs/cycle (retire 16 instrs/cycle)
Branch Predictor	8K-entry Combined, 8K entry gshare, 8K entry bimodal, 8K entry selector, 13 bits of history
Misprediction Penalty	10 cycles
Reorder Buffer	512 entries
Scheduler	64 entries
Functional Units	4 identical general purpose units
L1 I-Cache	32Kbytes, 4-way set assoc., 128 byte lines, 10 cycle miss
L1 D-Cache	32Kbytes, 4-way set assoc., 64 byte lines, 10 cycle miss
L2 Cache	512Kbytes, 8-way set assoc., 128 byte lines, 200 cycle miss penalty
Diverter Queue	128 entries
Spawn Latency	5 cycles
Inter-core Store-Load Forwarding Latency	5 cycles
Inter-core Register Comm. Latency	3 cycles per-hop

Table 5.1: Pipeline parameters.

Each point in the scatter plot represents the measured vs predicted improvement in performance (in cycles) for one task spawn option, averaged over many dynamic instances of that option for a 10Million instruction run. For the measured improvement, the system was allowed to spawn (multiple instances of) only one task choice in one run to isolate its impact from other choices. The total improvement over superscalar from that individual run is divided by the number of times the task was spawned to report the average improvement per spawned instance of that task. The model, on the other hand, can analyze all task choices in one pass.

These results show that the model predicts task performance accurately with relatively minor deviations from the expected value. Further, we find that the points with large deviations typically represent tasks that are spawned rarely, so warm-up effects for structures like caches, branch predictors, and data-dependence predictors introduce noise not captured by the model. In some cases (such as vortex), warm-up effects cause the model to under-predict. This is due to cache behavior, spawned tasks have four times the L1 cache available to them than superscalar execution and this can decrease capacity misses in some cases.

The results for each benchmark are accompanied by another set of graphs that show behavior of prediction error as a function of the number of instances of the task. These results show a clear trend of sharply dropping prediction errors as the number of instances of any task increase. Thus, warm up effects are amortized for more frequently spawned tasks. Thus, the model strikes a good balance between simplicity and accuracy.

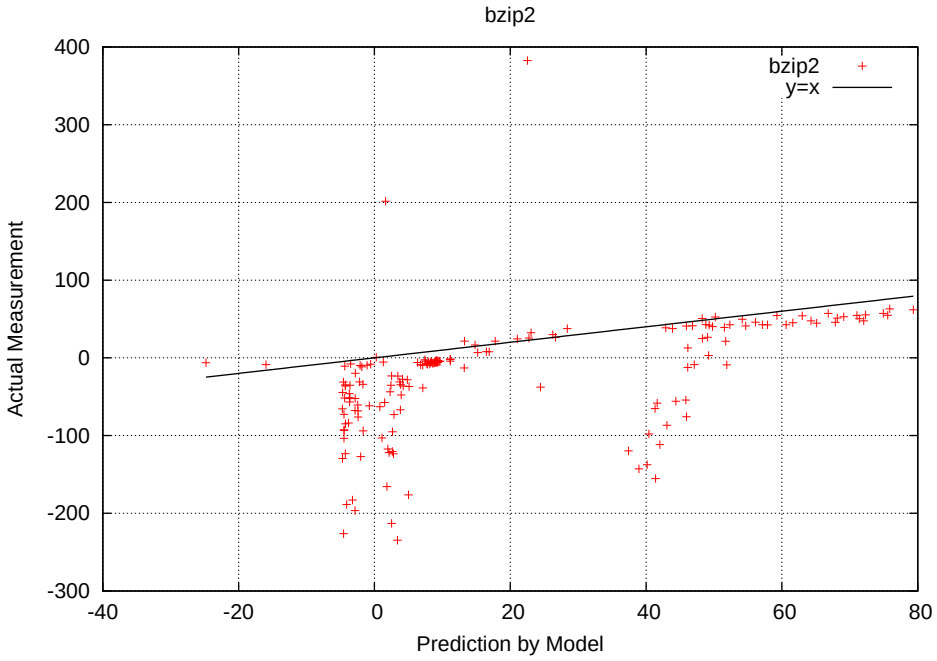

Figure 5.3: Bzip2: Prediction from the model compared to the measurements from a speculative parallelization system. The y=x line is shown for reference.

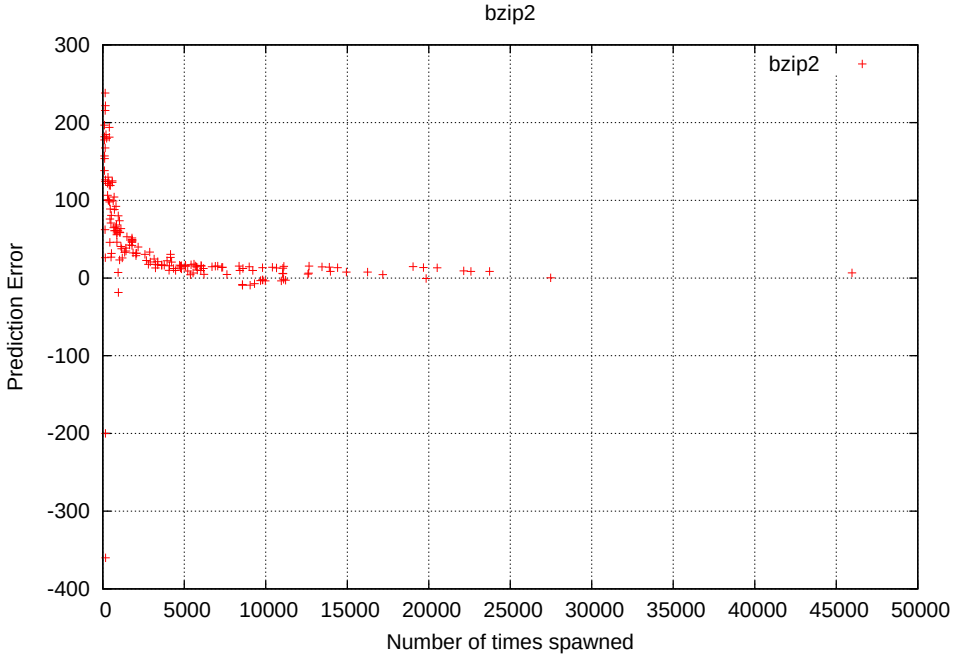

Figure 5.4: Bzip2: Prediction error as a function of number of times the task was spawned.

Figure 5.5: Validation for Bzip2

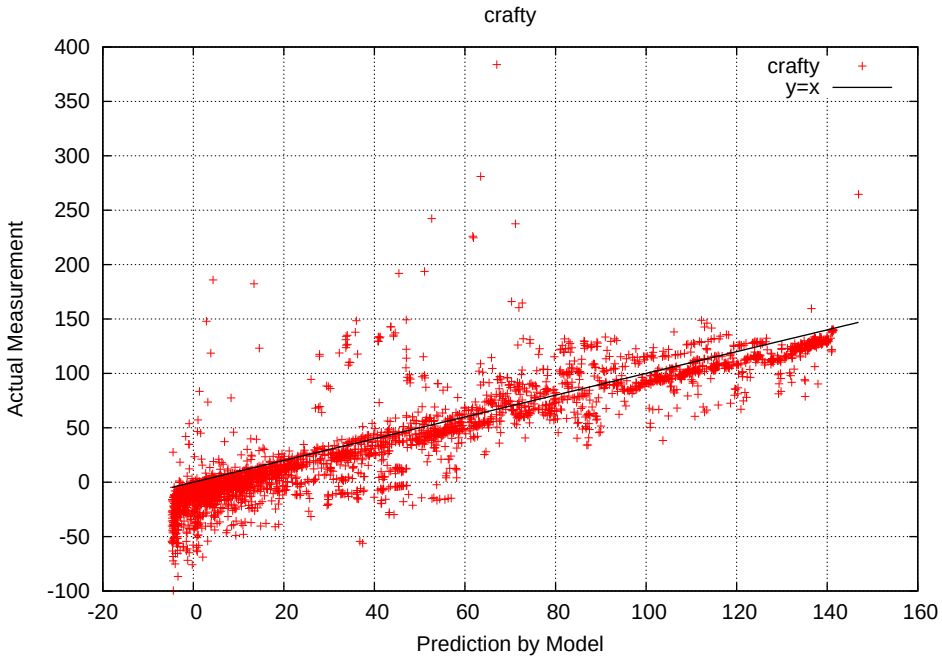

Figure 5.6: Crafty: Prediction from the model compared to the measurements from a speculative parallelization system. The y=x line is shown for reference.

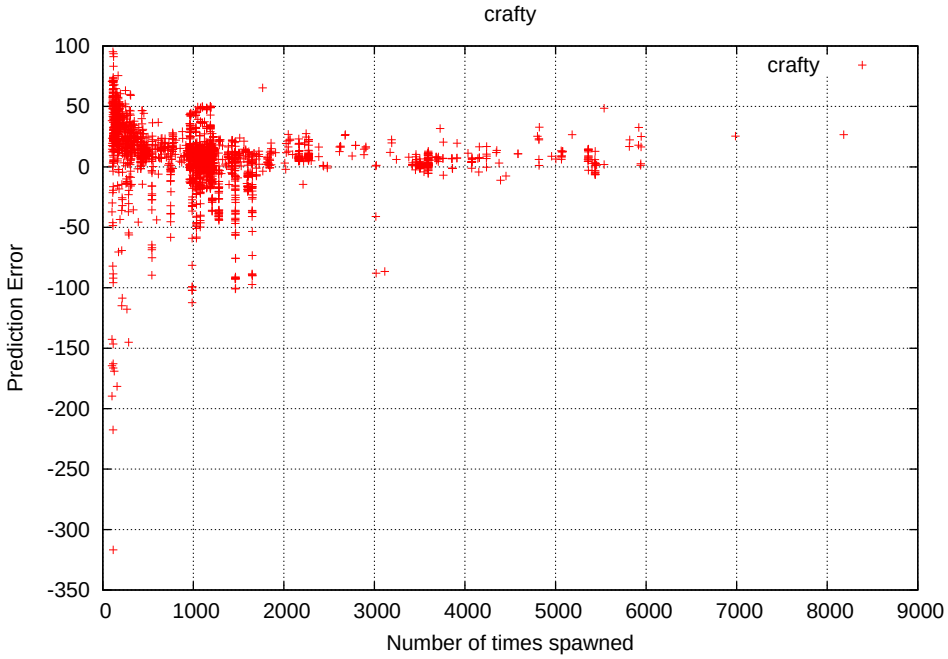

Figure 5.7: Crafty: Prediction error as a function of number of times the task was spawned.

Figure 5.8: Validation for Crafty.

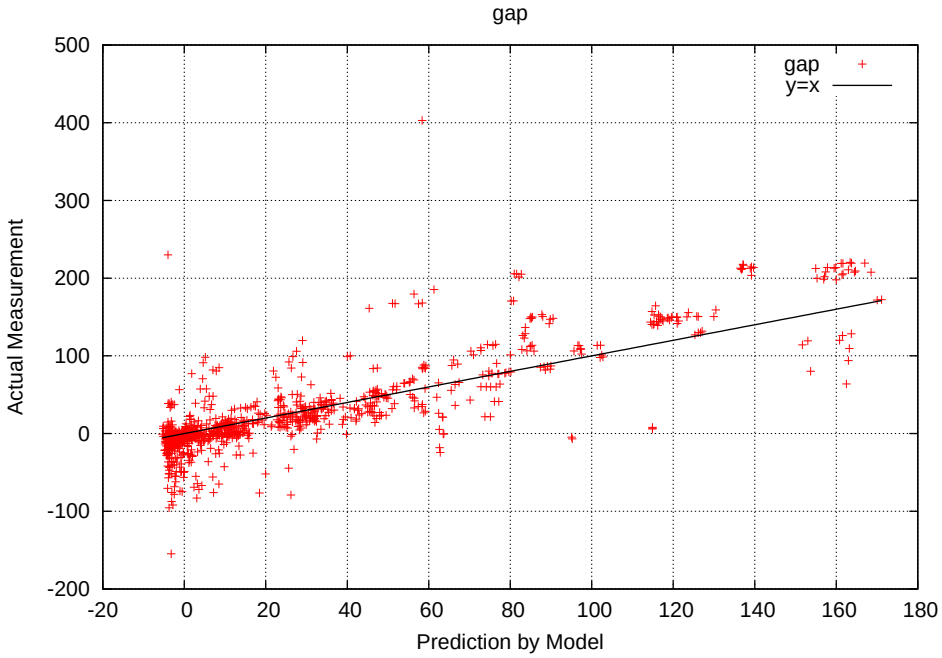

Figure 5.9: Gap: Prediction from the model compared to the measurements from a speculative parallelization system. The y=x line is shown for reference.

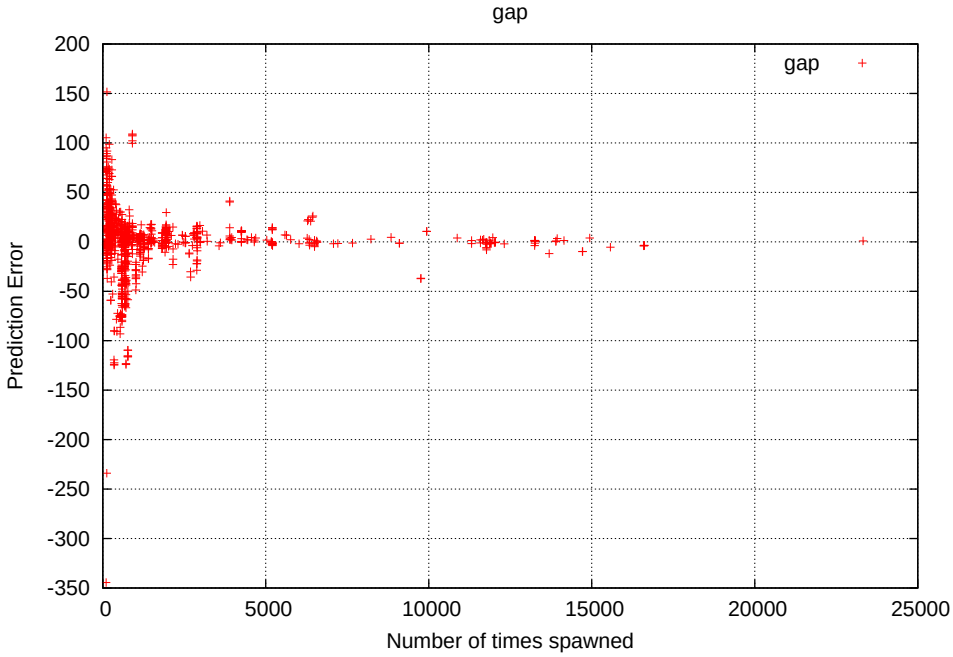

Figure 5.10: Gap: Prediction error as a function of number of times the task was spawned.

Figure 5.11: Validation for Gap.

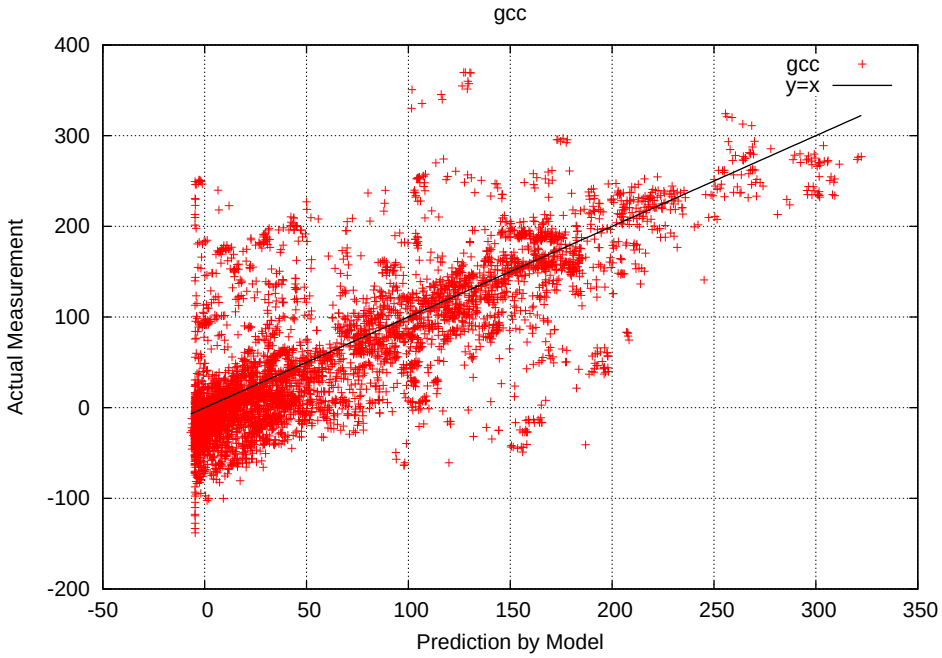

Figure 5.12: GCC: Prediction from the model compared to the measurements from a speculative parallelization system. The y=x line is shown for reference.

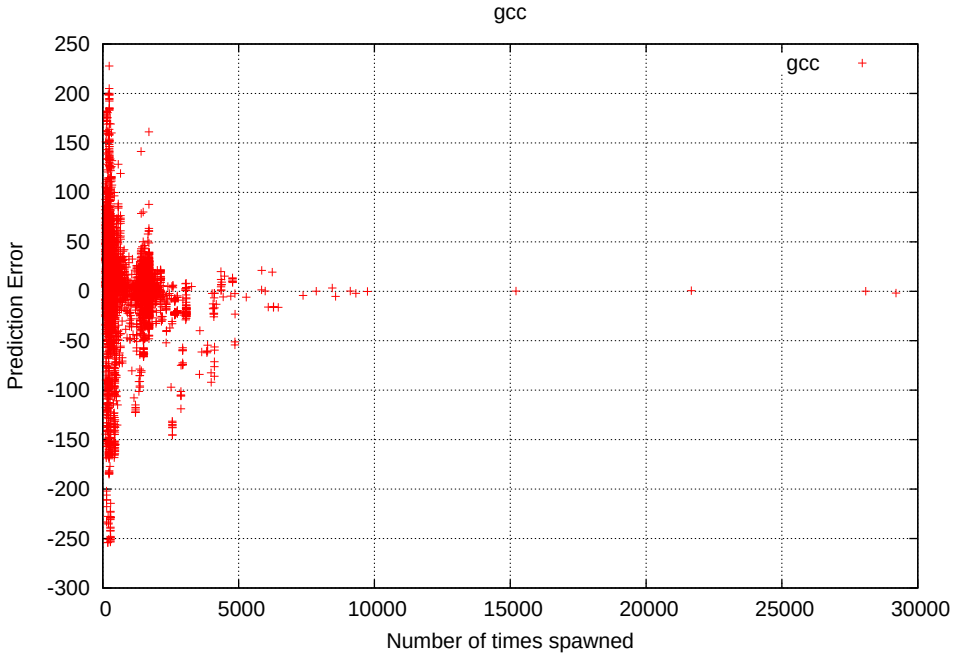

Figure 5.13: GCC: Prediction error as a function of number of times the task was spawned.

Figure 5.14: Validation for GCC.

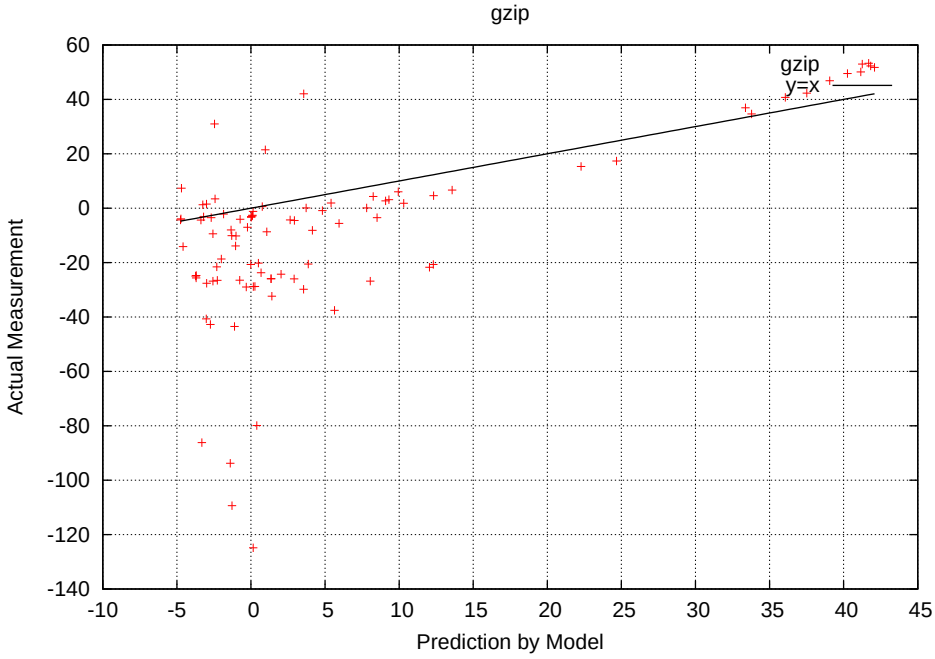

Figure 5.15: Gzip: Prediction from the model compared to the measurements from a speculative parallelization system. The y=x line is shown for reference.

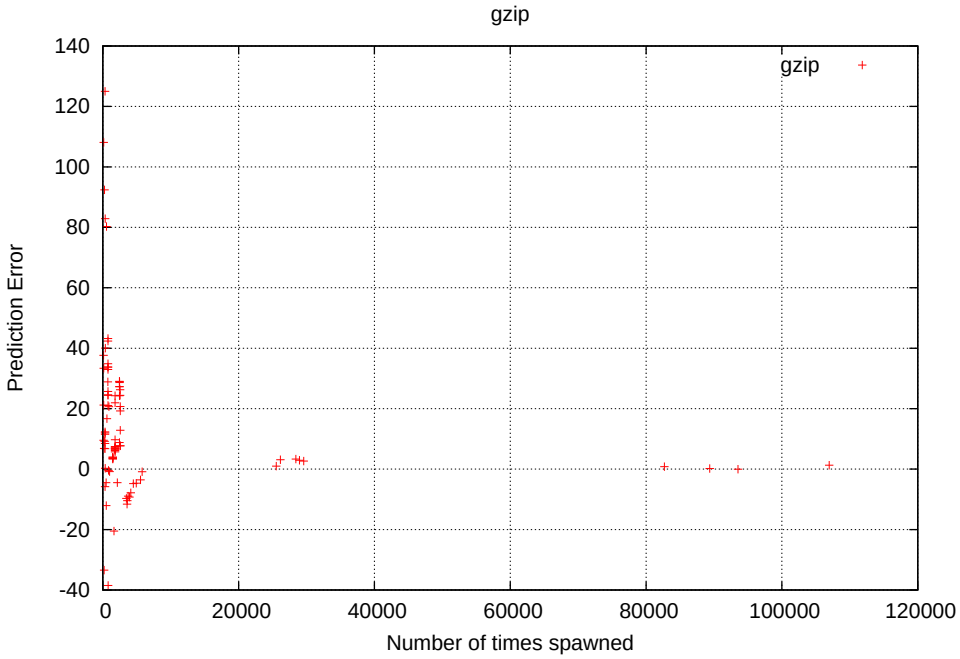

Figure 5.16: Gzip: Prediction error as a function of number of times the task was spawned.

Figure 5.17: Validation for Gzip.

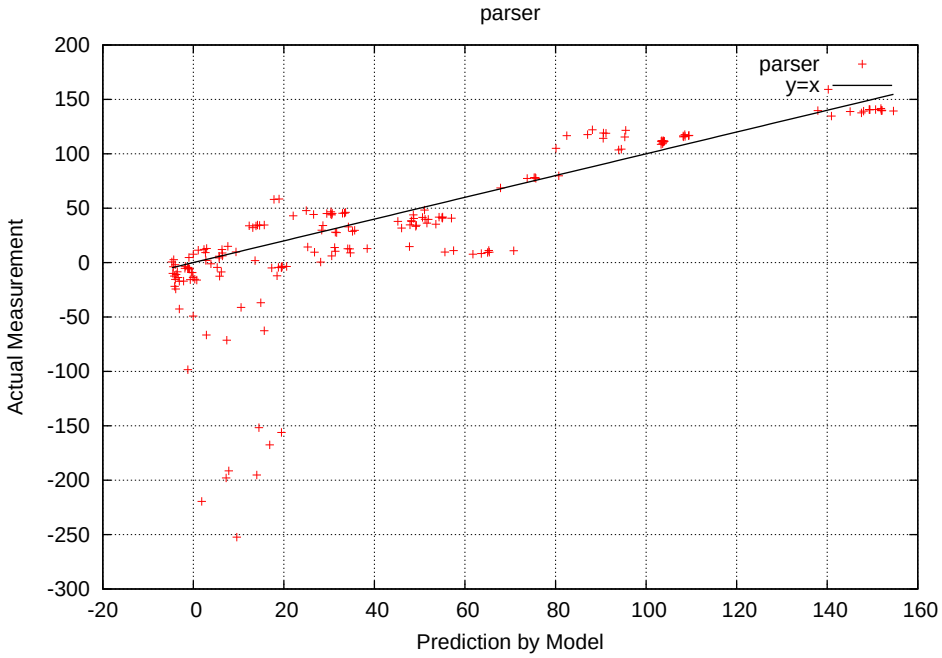

Figure 5.18: Parser: Prediction from the model compared to the measurements from a speculative parallelization system. The y=x line is shown for reference.

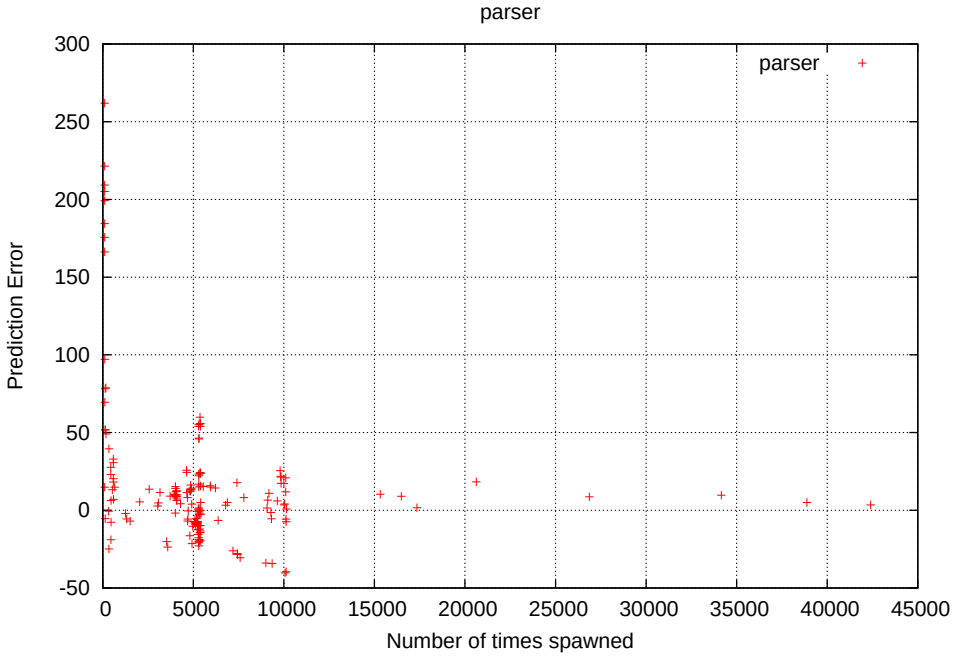

Figure 5.19: Parser: Prediction error as a function of number of times the task was spawned.

Figure 5.20: Validation for Parser.

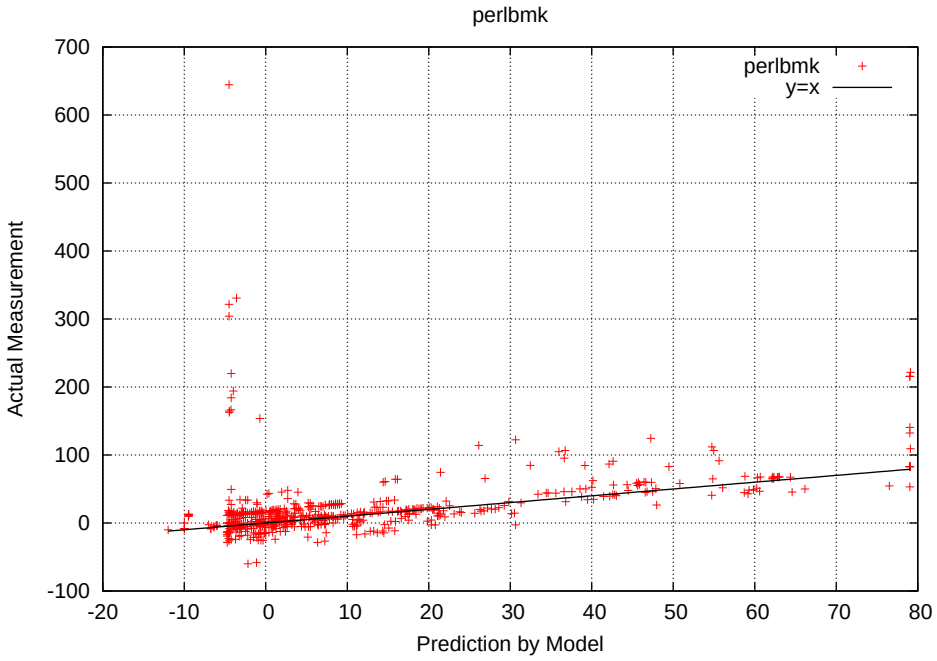

Figure 5.21: Perlbmk: Prediction from the model compared to the measurements from a speculative parallelization system. The y=x line is shown for reference.

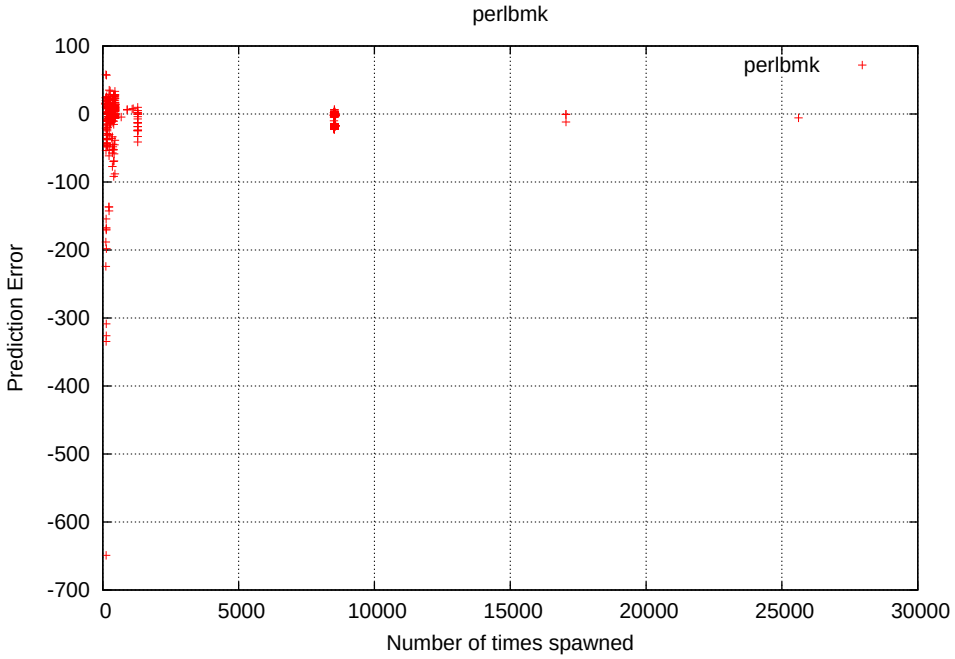

Figure 5.22: Perlbmk: Prediction error as a function of number of times the task was spawned.

Figure 5.23: Validation for Perlbmk.

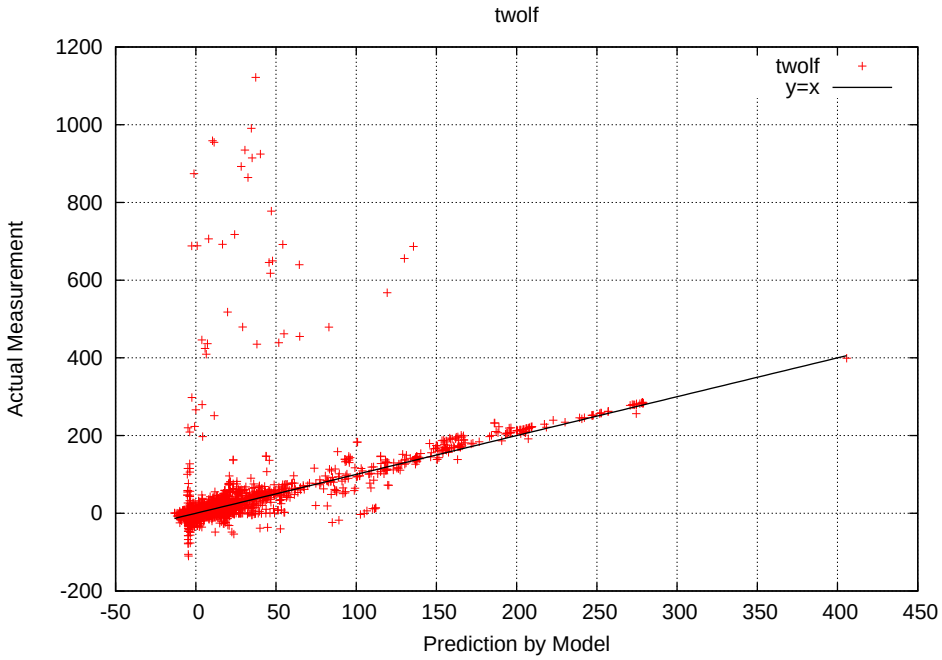

Figure 5.24: Twolf: Prediction from the model compared to the measurements from a speculative parallelization system. The y=x line is shown for reference.

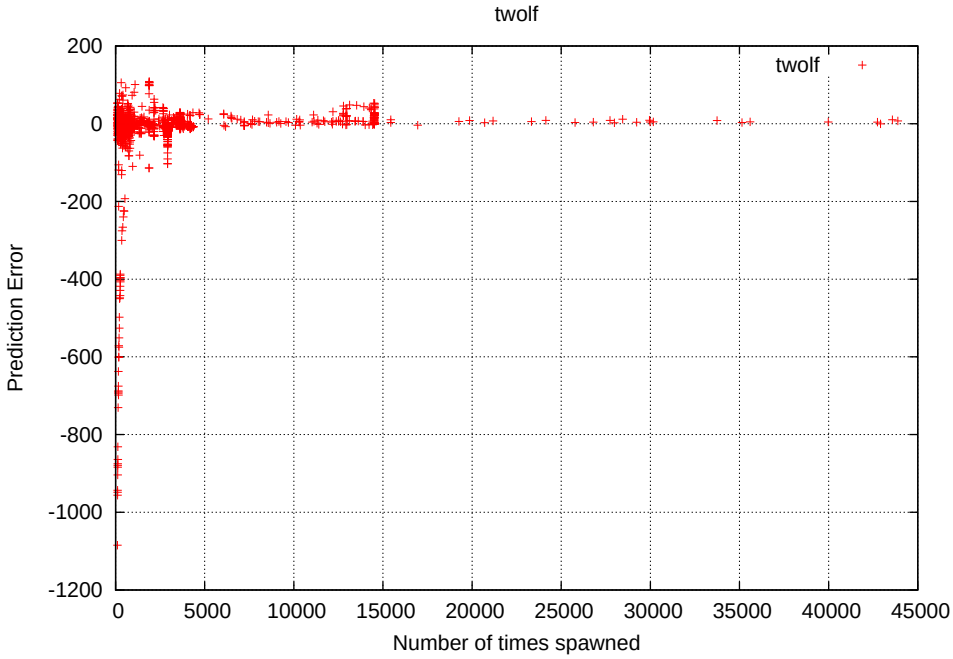

Figure 5.25: Twolf: Prediction error as a function of number of times the task was spawned.

Figure 5.26: Validation for Twolf.

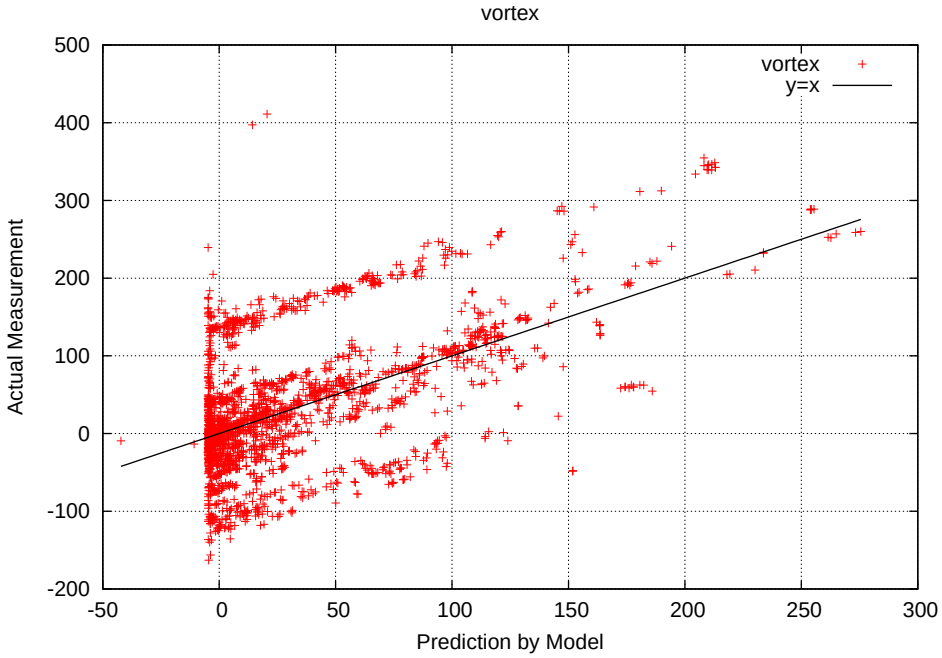

Figure 5.27: Vortex: Prediction from the model compared to the measurements from a speculative parallelization system. The y=x line is shown for reference.

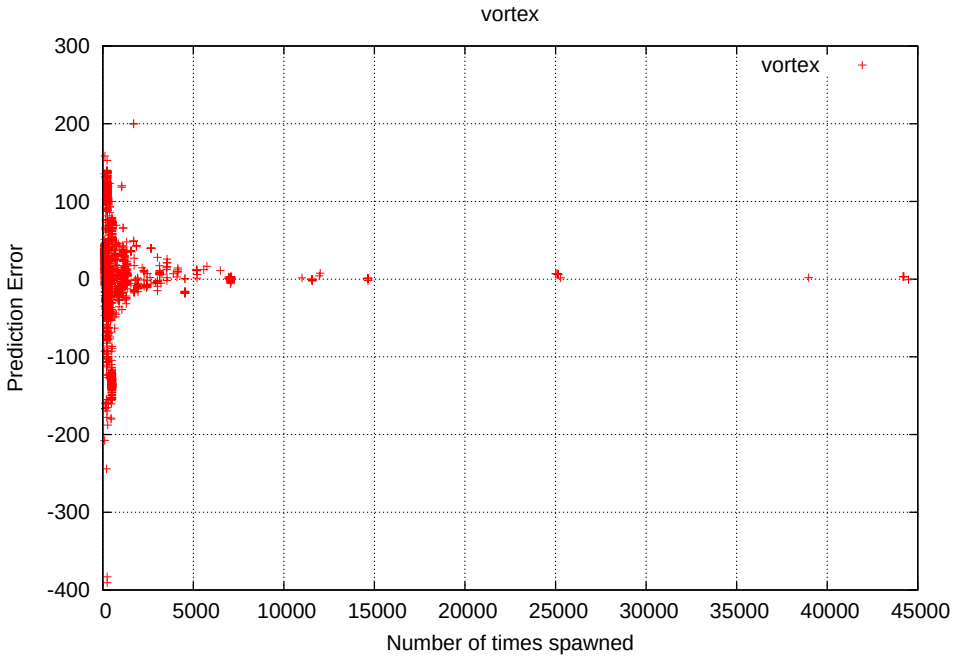

Figure 5.28: Vortex: Prediction error as a function of number of times the task was spawned.

Figure 5.29: Validation for Vortex.

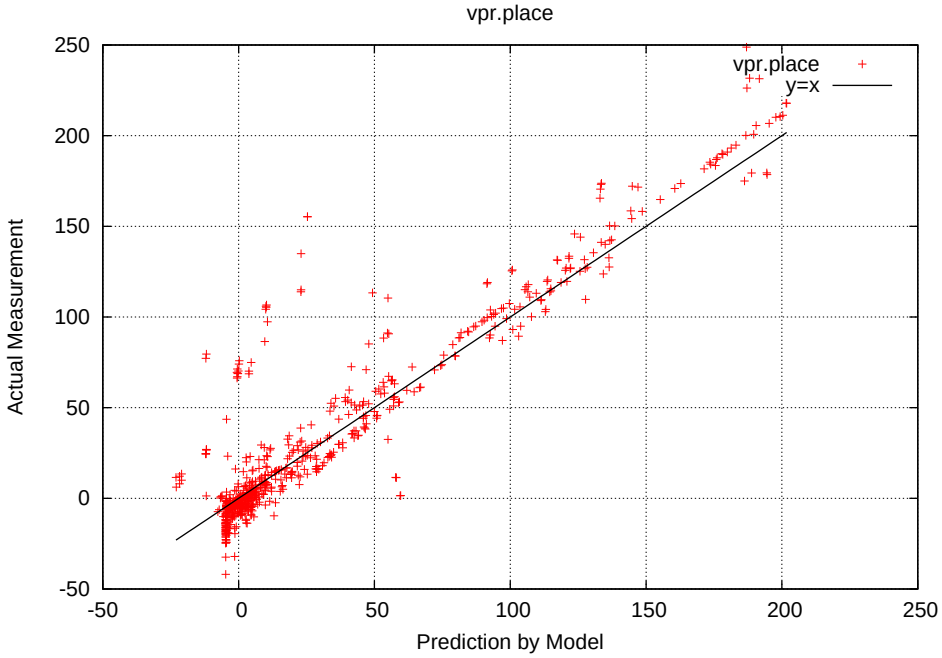

Figure 5.30: Vpr.place: Prediction from the model compared to the measurements from a speculative parallelization system. The y=x line is shown for reference.

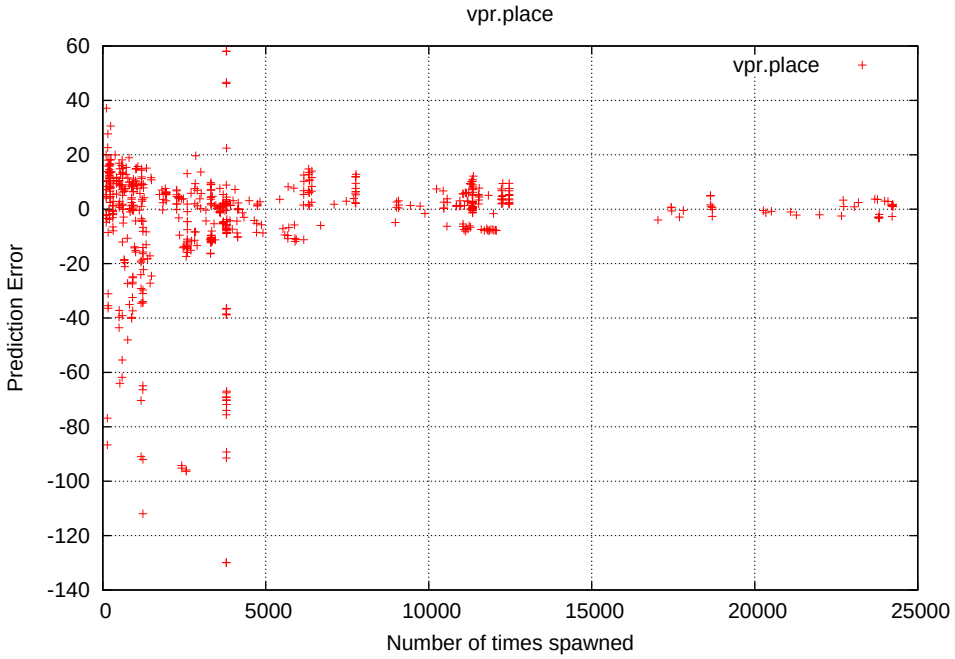

Figure 5.31: Vpr.place: Prediction error as a function of number of times the task was spawned.

Figure 5.32: Validation for VPR Place.

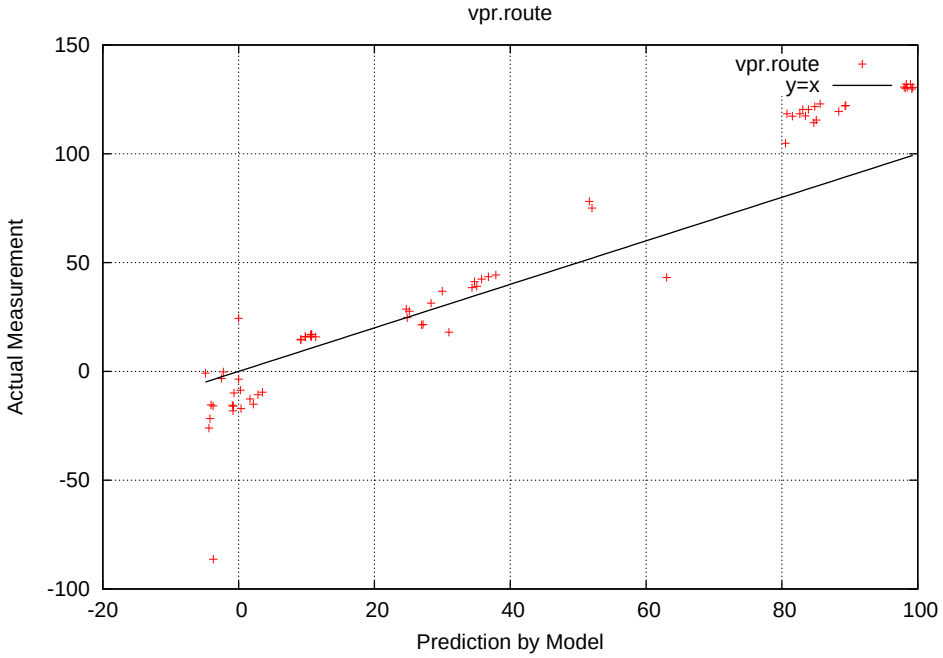

Figure 5.33: Vpr.route: Prediction from the model compared to the measurements from a speculative parallelization system. The y=x line is shown for reference.

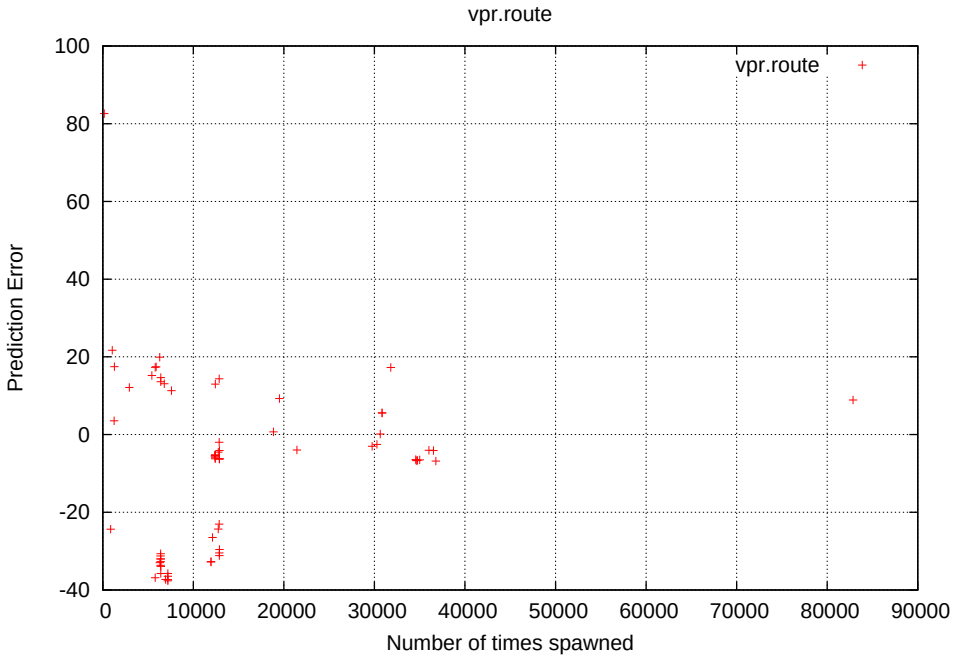

Figure 5.34: Vpr.route: Prediction error as a function of number of times the task was spawned.

Figure 5.35: Validation for VPR Route.

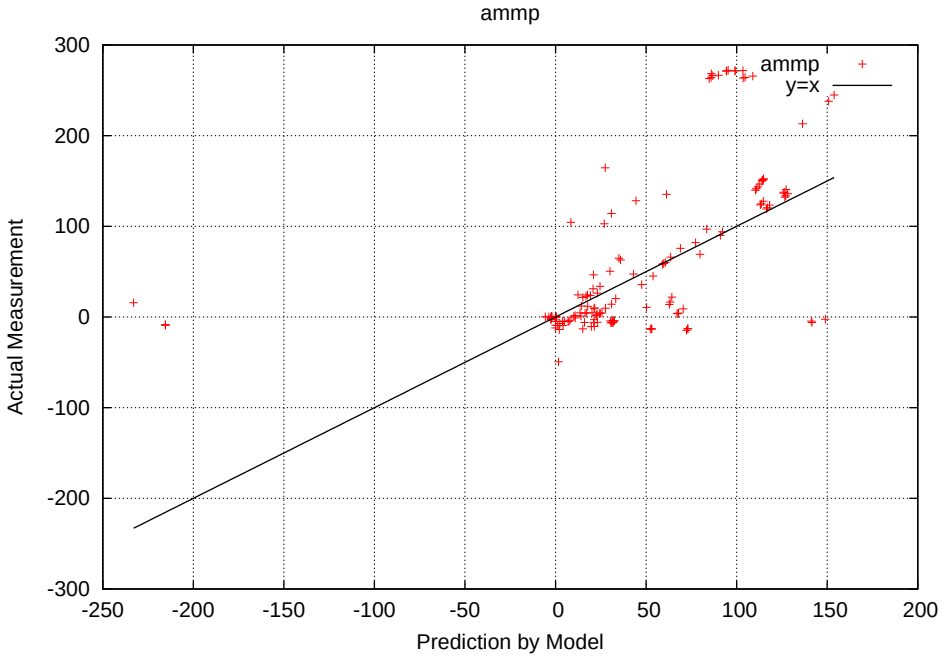

Figure 5.36: Ammp: Prediction from the model compared to the measurements from a speculative parallelization system. The y=x line is shown for reference.

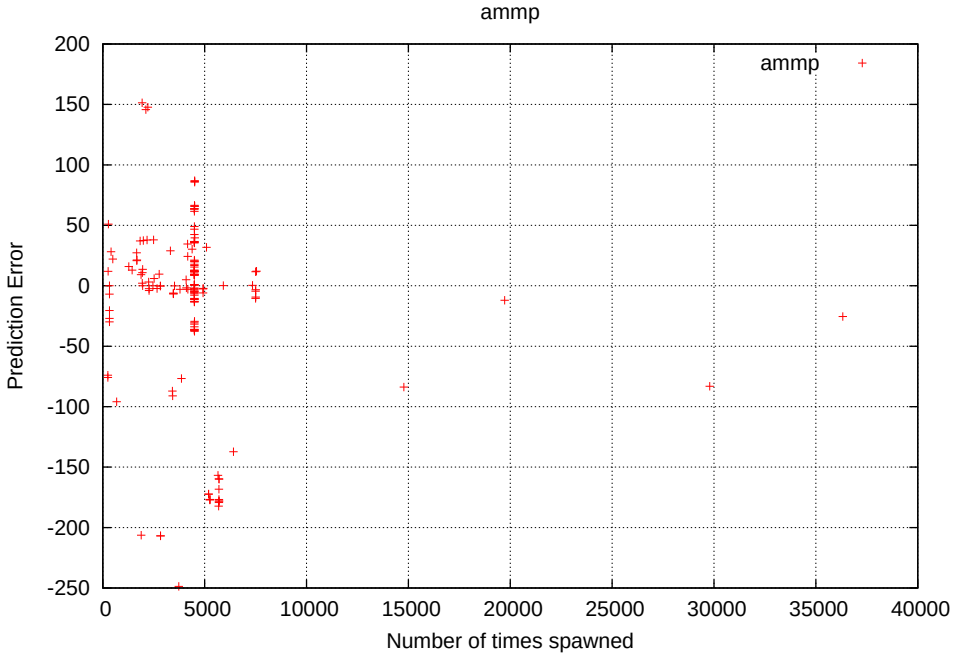

Figure 5.37: Ammp: Prediction error as a function of number of times the task was spawned.

Figure 5.38: Validation for Ammp.

PART II

EXTRACTING PARALLELISM ON POLYFLOW

CHAPTER 6

POLYFLOW: TARGET SPECULATIVE PARALLELIZATION SYSTEM

This chapter describes details of Polyflow, the target speculative parallelization system.

6.1 Terminology and High-Level Overview

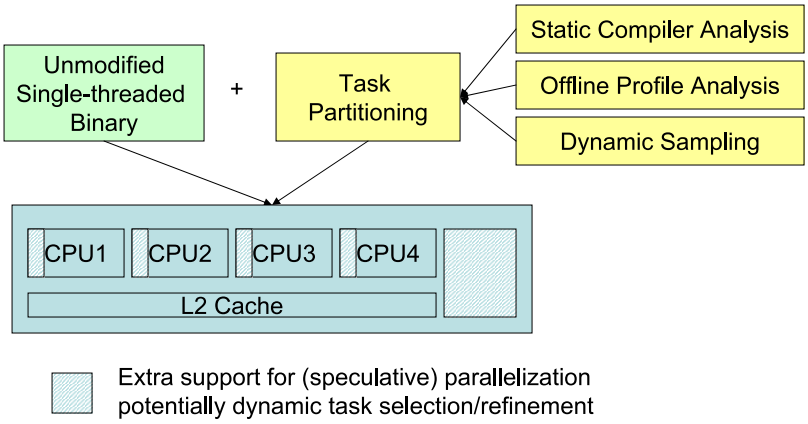

Figure 6.1: High-level setup for speculative parallelization.

Figure 6.1 illustrates the target setup for speculative parallelization. This thesis refers to this setup as the Polyflow architecture. It consists of a multicore architecture with additional architectural support for speculative tasking. The input is an unmodified single-threaded binary, with no special instructions for speculative parallelization. An optional input is task selection information that describes to the architecture how to partition the single-threaded execution into multiple (speculative) tasks. This task selection can be made in a variety of ways based on just compiler-based approach, or more input-dependent approaches such as profiling-based. In addition, the system might have dynamic support for identifying profitable tasks (or might rely solely on the task information provided as input).

Figure 6.2 gives an example of how Polyflow would speculatively parallelize a loop. The example shows the control-flow graph (CFG) of a loop, highlighting the loop body beginning at A, and the loop index update and branch at block B. The task information provided to the system is to spawn to block B whenever it encounters the beginning of loop body (instruction A). This thesis refers to A as the "**spawner**" instruction and B as the "**spawnee**" instruction.

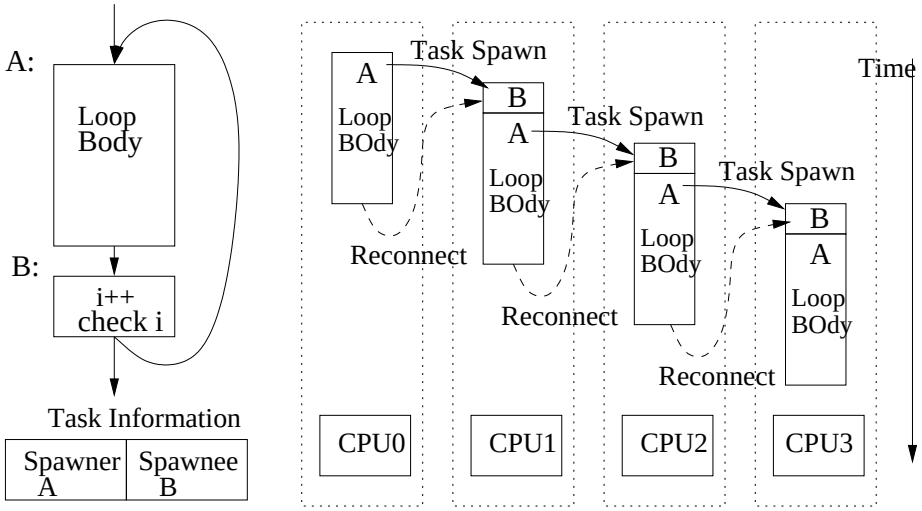

Figure 6.2: An example timeline for speculative parallelization.

The timeline shows how the execution unfolds on the system. When instruction A is encountered on core 0, it does a "**task spawn**" action, spawning instructions starting at block B as a separate task on core 1. The newly spawned task loops around (the loop condition evaluates to true in this case), and again spawns B when it reaches A. This process keeps repeating until all cores are used up, and resumes once cores become available for new tasks. If a core is not available when a spawner instruction is encountered, the task corresponding to that instruction is simply ignored (as opposed to some other systems where such tasks are buffered up and spawned when cores become free later on). The net result is that the execution of loop iterations is overlapped or parallelized (to the extent allowed by the architecture and program structure) for potentially higher performance. Once tasks have completed, they are merged through a "**task reconnect**" action. This preserves program semantics by presenting sequential behavior externally. Note that while figure 6.2 illustrates loop-based tasks, the system allows for irregular tasks as well. These tasks can spawn over procedure calls, hammocks, etc. Also note that this system does task spawn and reconnect actions "**in-order**". This means that once a task has spawned another future task, it can't spawn another task while the later task is live and has not been squashed. Task reconnections are carried out (and cores freed up) in the temporal order of tasks, that is the oldest task can reconnect to the task it had spawned, and only then can this merged task reconnect to the next oldest task. Later chapters will address the performance potential of out-of-order spawning [46].

The in-order spawning restriction lends itself well to a ring-like network between cores for task actions, since spawns and reconnections can only happen between adjacent cores. At any point, the oldest task is referred to as the **lead task** or the **non-speculative task** since its actions are not data-speculative in nature. Other tasks are speculative, with the youngest (and logically the most distant) task being the most speculative task.

6.2 Management of Data-Dependences

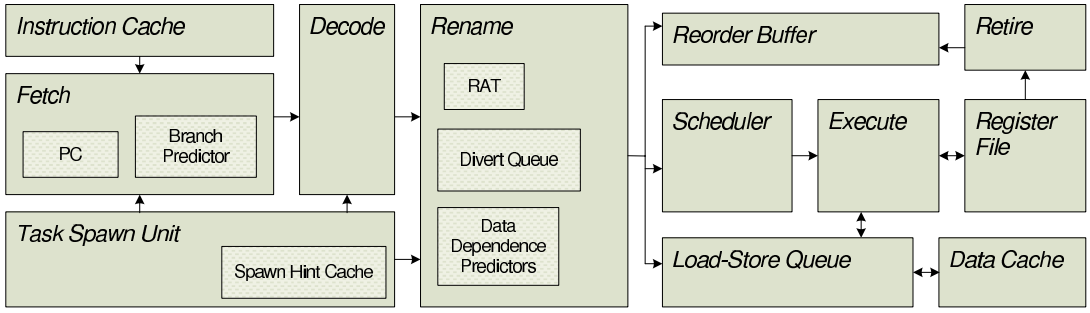

Figure 6.3: Pipeline of an individual Polyflow core.

There can be data dependences going from the spawner task to a speculative task spawned onto another core. These data-dependences need to be enforced to maintain correct execution. Polyflow enforces these data-dependences using the following techniques: speculation, synchronization and (in a very limited sense) value prediction. There can be two type of data-dependences: register-based and memory-based. The techniques used for the two cases are somewhat different.

6.2.1 Register Dependences

Register-based data dependences are somewhat easier to identify and enforce than memory dependences, as has also been observed by previous work. Register-based dependences can be unambiguously identified through a static analysis and can also easily be learned dynamically. Further, the limited number of registers in architectures makes tracking register-based dependences an easier task.

Polyflow's solution to enforcing inter-task register dependences is to identify instructions in the spawned task that might depend upon register values that will be produced in the spawner task, and make these (potentially) dependent instructions in the spanwed task wait until their value becomes available from the spawner task (or it is verified that the latest value was already available). In other words, Polyflow predicts inter-task register dependences, and synchronizes (potentially) dependent instructions on producers in earlier tasks.

Polyflow uses a dynamic structure in the architecture to train on the register-dependences between potential tasks. This structure, referred to as RSync [47] can train very quickly and can be used as a very accurate dependence predictor for future spawns of that task. Register dependence information is stored as a 32-bit (one bit for each register) vector for each task pair, with the bit for corresponding register set high if it was ever observed to be written by the spawner task, signifying that the spawned task may not have the latest value of the register and should "wait" on the value from the spawner task. Since the spawner task might itself be waiting on some register values from the task that spawned it, these registers are also marked as "waits-for registers" at task spawn time. If a register is written during task execution, its "waits-for" bit is cleared. In case a dependence

was not identified, a reconnection-time check signals the dependence violation which causes the violating task (and all later tasks) to be squashed. However, this is a rare action because the dependence predictor is very accurate.

6.2.2 Value-Prediction for Callee-Saved Register Dependences

Polyflow employs a very limited form of value prediction to break some register dependences. The specific case in which this is used is for callee-saved registers. These are the registers that must be saved by any procedure before they can be used, and restored to their previous value before returning from the procedure. The value stored in any such register is the same just after returning from a call as it was just before making the call, even though there might have been several inermediate writes to the register inside the called function. Polyflow leverages this insight to remove such “false” register dependences from the writes to a callee-saved register in a procedure to reads of that register after returning from that procedure if the spawned task jumps over the function call altogether [5].

6.2.3 Memory Dependences

Memory dependences are tricker to deal with than register dependences because they are harder to identify statically, involve more ambiguity than register dependences, and because there are many more memory locations than are registers. Polyflow uses a memory dependence predictor that is similar in nature to RSync, except that it uses store sets identifiers [48] rather than registers. The memory dependence predictor trains similar to RSync on memory accesses, and can generate a memory waits-for bit vector for a task pair. This can be used to synchronize loads/stores in the spawned task that access a particular store set.
However, a large number of memory accesses rarely/never alias to the same location even though they might access the same store set (which is a many-to-one mapping). In such cases, it might be better to speculate that no dependence exists and suffer the rare misspeculation penalty. Polyflow takes this approach for memory accesses, and if the observed dependence frequency is quite high for particular accesses, then tries to synchronize it based on the memory dependence prediction mechanism. This has been developed into an adaptive memory synchronizer, which adapts the behavior for each memory access. Further details can be found in Malik’s thesis [49].

6.3 Disambiguation of Memory Accesses and Forwarding of Data

Speculation on memory accesses can fail when a consuming load in a later task executes before it gets the correct data from the producing store in an earlier task. Such cases of failed speculation must be detected and the incorrect execution rolled back to ensure correctness. Speculative parallelization systems have a disambiguation mechanism to detect such violations. Polyflow’s disambiguation mechanisms have evolved over time. Initial proposals used load-store queue based

disambiguation, where each store broadcasts its address to other cores, and tasks on later cores can compare their loads to the store address to detect violations. Malik [49] has explored other complexity-effective solutions that enable disambiguation at load-retire time with a separate cache structure for disambiguation, inspired by Roth's approach for superscalar disambiguation [50].
A related issue is forwarding of memory data to later tasks for loads that are not synchronized but might still depend upon data from earlier tasks. Malik [49] refers to such loads as "lucky loads" because the timing of their execution works out so that they get correct data from producer threads even though they were not synchronized with their producers. Polyflow forwards memory data through a single, chip-level speculative cache. All stores write their value to this speculative cache upon execution, as proposed by Garg et al[51]. About 4% of dynamic loads receive their data from this cache, which takes an extra delay (e.g. 5 cycles).

6.4 Non-Blocking Scheduling through Divert Queues

Instructions that need to be synchronized to enforce inter-task data dependences must wait until data becomes available from producing task. This can incur considerable delay. Such synchronized instructions and their transitive dependents can meanwhile block the scheduler, preventing independent instructions in the task from executing. Polyflow addresses this concern by slicing out such "waits-for" and "transitively waits-for" instructions at the rename stage into a separate FIFO structure called divert queue. Only instructions whose dependences can be satisfied locally are allowed to proceed (with the exception of speculated loads or operations whose inter-task data was already available at spawn time). Instructions in the divert queue can be selectively "undiverted" as and when their producers release values.

6.5 Release Policy for Synchronized Instructions

As mentioned above, Polyflow tries to synchronize register-based dependences and frequent memory-based dependences that cross task boundary. Such dependent instructions are diverted and delayed until the producer value is released to the consumer instruction and is marked as ready to be scheduled. The policy that decides when this action happens is referred to as "release policy". Polyflow releases a value from a spawner task to the successor spawnee task when the following three conditions are met:

1. All branches in the spawner task have completed execution (thus Polyflow never releases data produced by bad-path instructions fetched due to misspeculated branches).

2. The spawner task has fetched and renamed (down the correct path) all instructions (in program order) prior to the first instruction in the spawnee task (ensures that the last writer has been seen).

3. The producer of the value has completed execution.

When the first two conditions are satisfied, the tasks are said to have reconnected. Therefore, the value is released when the spawner task has reconnected to the current task, and the producer of the value in the spawner task has also executed. If the dependence is from a task earlier than the spawner task, then all the tasks from the producer task leading up to the current task must have reconnected, and the producer instruction must also complete execution for the value to be released. Malik [52] has also explored more aggressive policies that try to release values earlier based on path confidence.

6.6 Task Spawn Management

The Task Spawn Unit (TSU) is responsible for managing tasking-related operations. The TSU stores task information in form of spawner-spawnee PC pairs in the spawn cache. Each cycle, the fetched PC is looked up in the spawn cache to see if it matches with one of the spawner PCs stored. If so, the TSU attempts to spawn the corresponding spawnee PC onto the next core if a task is not already running on it. Since Polyflow spawns tasks in-order, once a task has spawned another task on the next core, it cannot spawn another task in its lifetime unless the spawned task is squashed for some reason. If the next core is available, the TSU sends over a spawn command to the core. The spawn command contains the following:

- The start PC of the spawned task.
- Register dependence information that informs the spawned task which register values need to be synchronized. This is sent over as a 32-bit value as described above.
- Memory dependence information similar to the register dependence information.

Other actions also take place when a task is spawned. Branch history register for the spawned task must be initialized. Polyflow does this by clearing the global history register for the newly spawned task. Further, the spawned task needs values for the available registers. These are also sent over at the time the task is spawned.

CHAPTER 7

RELATED WORK IN SPECULATIVE PARALLELIZATION

This thesis builds on top of contributions and insights from a large amount of previous work in the area of speculative and automatic parallelization. This chapter summarizes representative work in some of the important directions in this domain.

7.1 Compiler-driven Automatic Parallelization

A lot of work was done in the 1980s and 1990s in compiler-based automatic parallelization of primarily scientific applications written in High-Performance Fortran. The focus was on parallelizing the main loops in the applications. The high-level approach was to do a static dependence analysis to construct the program dependence graph. Then, based on the result of this dependence analysis for a candidate loop, transformations could be applied to parallelize the loop. Hall et al. [53] describe a taxonomy of some of the transformations to parallelize loops. The transformations are divided into three categories: 1) Reordering: loop distribution, loop interchange, loop fusion, statement interchange, loop skewing, loop reversal; 2) Dependence breaking: privatization, array renaming, loop peeling, scalar expansion, loop splitting, loop alignment; 3) Memory optimizing: strip mining, scalar replacement, loop unrolling, unroll and jam; and 4) Miscellaneous: sequential $\leftrightarrow$ parallel, loop bounds adjusting, statement addition, statement deletion. Several of these transformations are described in detail by Kennedy et al. [54].
A major limitation to compiler-based automatic parallelization approach was the accuracy of static dependence analysis for memory-based dependences. Since memory accesses to different variables can alias to the same location, the compiler has to assume potential memory-based dependence between two memory accesses unless independence can be proved. This makes static memory dependence analysis very conservative and can prevent parallelization of several loops and other program regions where a dependence may not exist in practice even though independence cannot be proved.
Some projects sought to overcome this limitation through programmer feedback about potential dependences. ParaScope [53] is an interesting approach to an interactive but tool-aided program parallelization. The tool does a conservative data-dependence analysis on the candidate loop to be parallelized, and points out possible loop-carried data-dependences to the user. The user can reject these dependences if he/she knows for sure that a dependence does not exist. In addition, the tool allows the user to choose from a large set of transformations to parallelize the loop, which

significantly eases the task of loop parallelization.
SUIF Explorer [55] is a related approach to interactive program parallelization that combines the benefit of static interprocedural analysis with dynamic profile information. The tool consists of a "Loop Profile Analyzer" that points out the dominant loops in the application, and a "Dynamic Dependence Analyzer" that points out the data dependences that prevent parallelization. The focus is on breaking loop-carried dependences potentially with user help so that the loop becomes a DOALL loop. A slicing analysis can point out the statements that affect a particular conservatively identified data-dependence in question, so that the user can judge if the dependence actually exists and if it can be broken through techniques such as privatization.

7.2 Speculative Parallelization

Except for simple cases, compiler-based automatic parallelization is unable to make much headway due to the conservative nature of the dependence analysis. In particular, references to memory locations can alias for pointer variables etc. In those cases, it is very hard for the compiler to prove independence of memory accesses between proposed tasks/threads. This means that often the compiler can't parallelize many program regions because it is unable to prove data-independence, even though there may not have been an actual dependence between the regions.
This serves as the motivation for speculative parallelization. The key enabler is data speculation on ambiguous dependences. Data-speculation makes it possible to parallelize such regions where static approaches can't prove independence. If the dependence is unlikely, or infrequent, parallelization can be carried out speculating that no dependence exists. This is backed up with special hardware/software support that can detect if a "misspeculation" occurred, that is, there actually was a dependence that was not enforced. In such a case, there is a recovery mechanism, which might involve discarding all speculatively executed instructions and restarting execution from a correct program state.
Several research projects have explored speculative parallelization using data-speculation, also referred to as speculative multithreading or thread-level speculation (TLS). The Multiscalar project was a pioneer in hardware-based data speculation for parallelization [31]. Several other projects have explored this domain, including the CMU Stampede project [32], Stanford Hydra [33], Illinois TLS [34], Dynamic Multi-Threaded Processors project [35] and several others, for example [36–40].
Besides the above systems, there have been other flavors of speculative parallelization (SP) as well. Two of the important ones are control-independence processors and helper threading systems. The former class typically consists of a single fetch-unit that can leverage control-independence to make better use of the available fetch bandwidth and reduce the wastage from mispredicted branches. Skipper [27] and Transparent Control Independence [28] are two such systems. Both these systems need extra support for managing data-dependences correctly much like speculative parallelization systems. However, they are limited in scope compared to full-blown SP systems due

to the restriction of a single fetch-unit.
Another class of SP systems are those that pre-execute performance degrading events (PDEs) [56] in separate speculative threads, thereby improving performance by prefetching data or precomputing branch results [57]. These systems can spawn threads that consist of just the backward slice leading up to the PDE. Such "helper" threads can prefetch data for the main threads or precompute branch results, but don't actually commit any state and therefore improve performance through side-effects.

7.2.1 Challenges to Speculative Parallelization

There are numerous challenges to successful speculative parallelization. First, it requires extra architectural support for speculation which includes timely detection of misspeculation and support for roll-back to maintain correct execution. Thus, it requires extra silicon area (and associated costs in circuit complexity and extra power consumption). Much research has been done to develop architectural mechanisms for speculative parallelization that minimize the impact on power consumption, circuit complexity, and impact on circuit critical path.
Another challenge is maintaining correctness of execution by preserving sequential semantics externally even though the application is speculatively parallelized internally. The key impediment to this requirement is the presence of data-dependences between the speculative tasks that must be enforced. If dependences are frequent and predictable, they can be synchronized with hardware (and/or compiler support) support [47]. Otherwise, systems can speculate that there was no dependence, and roll-back upon detecting a dependence that was not enforced. Based upon the nature of the dependence, one mechanism might suffer lower cost than the other. Future (in program order) speculative tasks might need values computed by earlier threads, and this might need extra communication channels such as a separate inter-core operand bus.
Finally, the above architectural techniques introduce costs to spawning a future region as a speculative task, in the form of penalties for communicating values to the task, the extra delay incurred by synchronized dependences compared to single-threaded execution, penalty incurred by misspeculated dependences, and other overheads associated with speculative tasking. Therefore, another major challenge is deciding how best to partition the application into speculative tasks to maximize the achieved parallel performance. Tasks that expose a lot of parallelism while incurring low cost can lead to large increases in performance. Conversely, tasks that incur large costs/penalties without yielding much parallelism can slow performance down.

7.3 Task Selection for Speculative Parallelization

The policy that decides which tasks are spawned for speculative parallelization is referred to as the "task selection" policy, and constitutes a major chunk of this thesis. Designing a good task selection policy is a challenging problem because there are typically a large set of task choices available for any given application, but only a few of them might be profitable to performance. The

source of profitability can vary because parallelism can come from cache misses, mispredicted branches, independent instructions, etc. Parallelism can come from many sources and can come from regular tasks such as loop iterations, or might be irregular in nature. Finally, the profitability of a task also depends upon the cost imposed by data dependences and other tasking actions.
Task selection for speculative parallelization has been approached in multiple ways, varying from compiler identification of tasks, to dynamic heuristic-based task creation. This section summarizes some of the representative related work in this area. There are two axes along which the related work can be classified. First is the set of potential task choices that systems consider for speculative parallelization. The second is the way in which systems do a cost-benefit analysis on the available task choices, and select those that provide large benefits, and suffer from minimal costs.

7.3.1 Potential Task Choices Considered

There are mainly two schools of thought in this direction. One set of systems rely on regular tasks based on program constructs like loops and procedure calls. The benefit of this approach is that these tasks are easy to identify, lead to a reasonable number of tasks that can be analyzed well, and hopefully lead to good coverage of program execution. The other approach is to allow a larger set of tasks that might be irregular in nature. The benefit of this approach is greater flexibility that enables successful parallelization even if parallelism cannot be found through regular tasks. This section describes the approach that several systems have taken.
Many compilers for thread-level speculation (TLS) rely on loops as candidates for parallel execution, and loop iterations are the only possible tasks. Loop unrolling, and loop interchange are applied in conjunction with task selection to create tasks of suitable sizes. This includes the STAMPEDE system [58, 59] which focusses on loops that provide high program coverage, and considers different unrolled versions of such loops as possible tasks. Several other systems also focus solely on loop iterations as speculative tasks, such as the TEST system [60], Du et. al [61], Wang et. al [62], and many others [40, 59, 63]. The clustered speculative multithreaded processor used a dynamic loop detector to identify and spawn loop iterations as tasks [64]. Such loop-iteration based spawns were found to be preferable to loop fall through or procedure fall through spawns in the context of the clustered speculative multithreaded processors [65].
Some systems consider, in addition to loop iterations, tasks that include loop continuations, procedure calls and procedure fall-throughs [43, 66]. The Dynamic Multi-Threading (DMT) processor [35] uses dynamic heuristics to spawn at procedure and loop fall-throughs. It approximates loop fall-throughs by spawning the static address directly following each backward branch. A history buffer is used to predict after-loop thread addresses that differ from this default value. A subsequent work [67] implements a run-ahead policy that also spawns the instruction following an L3 cache miss.
However, for several applications, regular tasks derived from loops and procedure calls may not be enough, because parallelism might be more irregular in nature and there might be complex control-flow involved. A broader set of tasks might need to be considered. One of the contributions

of this thesis is a demonstration that tasks derived from postdominator analysis subsume heuristics like loop and procedure spawns, but also provide a variety of other task options that are important for performance. But there have been previous systems that have considered a broad set of tasks. The Multiscalar compiler [68, 69] allowed a region of the Control-Flow Graph (CFG) to be arbitrarily partitioned into two tasks through a cut. The only restriction being that each task had a single entry point, and all basic blocks within a task were connected. While this allowed for general tasks, it also introduced the potential for inter-task control misspeculation since a task could have several successors and spawning a task required speculating that it would be reached in future. There are ways of reducing the control misspeculation penalty. SPSM [70] was an early speculative multithreading system that considered tasks that are control equivalent to their spawner for starting new tasks. Control equivalence means that tasks are control non-speculative with respect to their spawner. Other systems try to use profiled path information to create speculative tasks along the frequently executed paths. Control Quasi-Independence (CQIP) uses profile information to reconstruct the dynamic program control flow graph with edges weighted by execution frequency [71]. Basic block pairs that are likely to lie on the same path are identified as possible spawning point and control quasi-independent points. Bhowmik et al. [72] also describe a compiler system that uses path profiles to identify tasks. It starts out by trying to create tasks out of loop iterations. Next, it tries to create tasks along common paths, as well as infrequent paths, for each immediate postdominator pair.
The Skipper [27] processor exploits control independence to skip instructions control dependent on hard to predict branches. When it encounters a low-confidence branch, it skips the region control-dependent upon the branch, and instead fetches and executes instruction control independent of those branches. In a sense, it “spawns” the closest postdominator of the branch, although this is done for a single-fetch unit processor.

7.3.2 Heuristics to Estimate Task Benefit

The other important aspect of a task selection policy is the strategy to make a selection from the available choices. A large number of task choices might be available to a speculative parallelization system. However tasks can compete with each other. That is, spawning one task might preclude spawning a set of tasks because it might overlap with these tasks (partially or completely). Further, even non-overlapping tasks can compete for limited thread resources. It is the job of the task selection policy to make a selection that maximizes the achieved performance given the available resources.
This is a hard problem because it is not straightforward to estimate the impact of a given task choice on performance when it is spawned. A variety of sources of parallelism and costs can affect individual task profitability. The other consideration to be taken into account is how do a set of tasks affect each other with regards to resource contention as well as other factors like penalties, etc. This section summarizes how different projects approach task selection.
The STAMPEDE TLS system [58] shortlists a set of loops, and considers different unrolled

versions of these loops as potential tasks. In order to understand which tasks are profitable, it runs each potential task in isolation on the detailed TLS model to measure improvement from each possible version. The best unrolled version is selected for each loop spawn point. However, this approach is unlikely to be profitable where a large number of task choices are available, and so other systems have explored heuristics to estimate task profitability.

Another somewhat more sophisticated trace-analysis technique is used by the Mitosis compiler [73]. It builds upon the idea of Control-Quasi Independence [71] to select tasks, while ensuring a minimum task length. One of the novel aspects of the Mitosis system is that it tries to avoid costly inter-task data communication by generating precomputation slices (or p-slices) for each task to compute live-in values. It uses extensive CFG and Data-edge profiling to identify live-ins for p-slices. Next, it has a selection phase that runs on synthetic traces. The phase operates on a given subroutine (and loop level) at a time, and makes a selection for that level, and this step is repeated from the innermost to the outermost subroutine. Within each step, a greedy selection heuristic expands the set of selected tasks by iteratively picking the task that maximally improves performance over current selection until no further improvement is achieved. To evaluate the performance for a given selection, it "simulates" speculative parallelization on the synthetic trace. However, the simulation is done on an abstract model of the system modeling few architectural details, for example assuming that each instruction takes unit time, and therefore can be made faster than a more detailed model.

Other systems use profiling information about task size and dependences and use heuristics to make their selection. The Multiscalar compiler [68, 69] identifies the following as main costs to speculative parallelization: control flow speculation, data communication, data-dependence speculation, load imbalance, and task overhead. These costs are incorporated in a selection heuristic that performs task selection by walking the static program control-flow graph (CFG) and partitioning it into tasks. There are three heuristics: task size, inter-task control flow, and inter-task data dependences. The task size heuristic uses loop unrolling and function inlining to make tasks of appropriate size, and thus minimize load-imbalance. Tasks are not allowed to cross loop or function entries or exits. The control-flow heuristic limits the number of successors of a task to reduce cost of inter-task control misspeculation. The data-dependence heuristic tries to place producer and consumers of frequent dependences within the same task. A later related work [74] annotates the static CFG with edge weights that combine the impact of load imbalance, data-dependence cost as well as control prediction penalty into one single metric, thereby giving equal consideration to all three. The min-cut algorithm is then used to best partition the CFG into tasks.

The factors identified above are indeed the most important considerations in task selection. The inter-task control-misspeculation aspect is somewhat specific to Multiscalar due to the extra flexibility it allows in terms of task structure. Several other systems focus on just load imbalance and data-dependences. For example, Du et. al [61] focus on minimizing misspeculation cost due to data-dependences in spawned tasks. They construct a control-data flow graph where

data-dependence edges are annotated with profiled dependence probabilities. This helps estimate the data misspeculation cost for a given loop spawn option (their system doesn't synchronize memory dependences). The selection component uses the estimated misspeculation cost and task size for admittance.
Similarly, Wang et. al [62] profile to estimate the probability that the spawned task will suffer a data misspeculation and the cost of this misspeculation. Profiling is also used to estimate communication and synchronization delays. These are used for task selection, along with task size information. An interesting aspect is that they construct a loop graph that captures loop nesting relationships. The loop nest that maximizes parallel performance is selected for spawning.
Bhowmik et al. [72] also follow a similar approach that considers task size and cost of data-dependences for task selection. They have two heuristics to estimate the cost of data-dependence: Data-Dependence Count (DDC), and Data-Dependence Distance (DDD). The DDD heuristic is similar to the synchronization delay heuristic in that it estimates the minimum stall time of the consumer instruction due to producer in the previous task. The DDC heuristic, on the other hand, counts the number of data-dependences that cross the task boundary, with lower weight given to distant dependences.
The TEST system [60] also relies on task size and data-dependence cost to make its selection. An interesting aspect is that it provides dynamic task selection support in the Java Virtual Machine, by profiling prospective loops (referred to as Speculative Thread Loops or STLs in the study) to select the most profitable loops for speculative parallelization. The profiler does two analyses to quantify the potential of a prospective loop: load dependency analysis, that tries to capture the impact of inter-task store-to-load dependencies on performance, and speculative state overflow analysis, which checks that the task is of appropriate granularity and won't overflow speculative buffers. Loops that provide good coverage and meet thresholds on above metrics are selected.
Other systems realize that there are other factors besides task size that determine task profitability. The POSH compiler [43] profiles to measure/estimate the following information for each task choice: the number of instructions in the spawned task that overlap the spawner task, wastage due to squashes resulting from dependence violations, and prefetching benefits due to original cache misses that were fetched earlier due to the spawn. These are combined into a single "benefit" metric that is used for admittance. The compiler also tries to hoist loop, loop fall-through, and procedure fall-through spawn points as high as possible, using control equivalence along with other constraints on data dependence and task spawn ordering.

7.4 Program Transformations for Speculative Parallelizability

7.4.1 Speculative Program Transformations

Static or run-time approaches that take a given application binary and try to best partition it into tasks are constrained because they have to preserve application semantics. However, there can be some flexibility in terms of recompiling the application if it leads to significant improvements in

parallelizability. Non-speculative loop transformations like unrolling, distribution, interchange, etc. are applicable in general and also to speculative parallelization. However, support for speculation enables another set of "speculative transformations" that get high performance in the common case, but can catch violations in the rare case to maintain correctness. Two main projects have explored this domain.

Vachharajani et al. [75] propose Speculative Decoupled Software Pipelining (SpecDSWP) as a technique to parallelize loops that contain dependences and recurrences, with the help of control- and data-speculation. The work is motivated by the concepts of software pipelining. The program dependence graph (PDG) of the candidate loop is divided up into multiple strongly connected components (SCCs), such that there are no cyclic dependences between any SCCs. These components are then run as separate tasks. Dependences that are speculated upon can be removed from the PDG to parallelize many loops. Queues are maintained to buffer inter-task data communication.

Zhong et al. [76] present speculative transformations that can enable speculative parallelization of a large number of loops. These transformations are adaptations of counterparts from the domain of automatic (non-speculative) parallelization of Fortran programs. The transformations evaluated include speculative loop fission, prematerialization, infrequent dependence isolation, variable privatization, reduction variable expansion, and ignoring long-distance memory dependences.

7.4.2 Revisiting Application Implementation

Often there are major roadblocks to parallelization imposed by the way the application is written, and parallelization is prevented by the need to preserve application semantics. Researchers have explored the scope for parallelization by minor refactoring of applications (and potentially relaxing application semantics) targeted towards speculative parallelization.

Prabhu et al.[77] explored the potential of thread level speculation for several SPEC2000 applications by manually applying transformations such as parallel reductions, loop slicing, etc where applicable. They also explored advanced value prediction techniques to reduce the cost of data dependences. They observe high costs from managing inter-task data dependences and communication, and the overheads of speculative parallelization.

Bridges et al.[78] show that the upside potential of speculative parallelization can be enhanced by compromising on sequential semantics. They identify places in the code where they can place annotations that specify legal transformations, such as reordering multiple invocations of a function with respect to each other (commutative) and sacrificing the quality of result (e.g. compression ratio) for parallel performance. However, their modeled architecture is not restrictive in terms of task sizes, cost of inter-task data-dependences, etc. Our study explores the costs imposed by several such constraints to understand the performance potentials and bottlenecks for different architectural choices. Our parallelization maintains sequential semantics.

CHAPTER 8

TASK SELECTION FOR POLYFLOW

Chapter 5 developed a criticality-driven model of task behavior. This chapter shows one application of that model to make task selection for speculative parallelization on Polyflow architecture. The chapter also compares the performance of that task selection policy against other selection heuristics from literature, and shows that the model enables the design of a superior policy.

8.1 Comparison Policies

Section 7.3 described a variety of task selection strategies used in several speculative parallelization systems. Doing a fair and quantitative comparison against each one of those policies is tough because each system has its own nuances and effects that dominate the behavior. For example, systems that suffer large amounts of memory misspeculations need to minimize that to be profitable and therefore most of the focus is on selecting tasks that minimize the chance of misspeculations. Systems that don't slice out instructions dependent on inter-task dataflow block later independent instructions and therefore suffer a large synchronization cost whenever any data-dependence crosses task boundary. Polyflow tries to minimize inter-task data misspeculations by synchronizing on frequent dependences (section 6.2) and implements a non-blocking scheduler as described in section 6.4. Therefore policies that worked well on some other systems may not be as useful on Polyflow because of the underlying architectural techniques in play.
With these points in mind, this section tries to adapt some task selection policies in previous work for Polyflow. Section 8.1.1 develops a policy based on the insights of Skipper and DMT systems. Then section 8.1.2 develops a policy based on Multiscalar task selection policy, but this also captures the insights from other systems such as Johnson et al.[74], Du et. al [61], and the Data-Dependence Count (DDC) heuristic of Bhowmik et al. [72].

8.1.1 Closest Spawn Policy

The first comparison policy used tries to approximate the task spawn policy used in Skipper [27] and DMT [35] architectures. The Skipper architecture, upon encountering a low-confidence branch, spawns the closest control-independent point of that low-confidence branch as a task that runs on a separate hardware context. The DMT architecture spawns the fall-throughs of loop

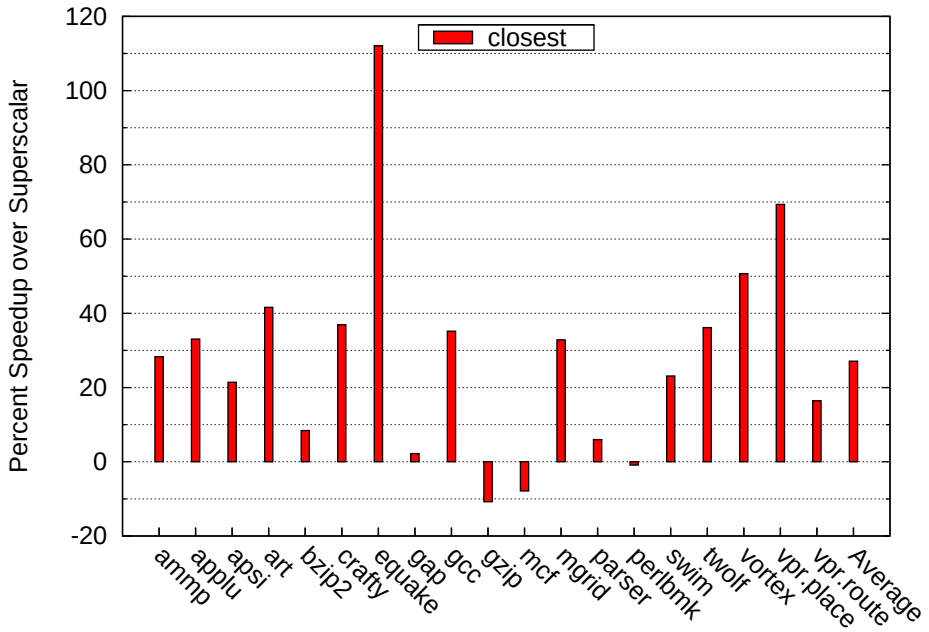

Figure 8.1: Performance of a speculative parallelization system that spawns the closest available control-independent point.

branches and procedure calls as a separate task.

The policy implemented here subsumes the two policies mentioned above. It tries to spawns the closest (immediate) postdominator as a separate task. Thus it captures the closest postdominators of low-confidence branches (Skipper policy). In addition, fall-throughs of procedures and loops are the immediate postdominators of the loop branch and procedure call respectively. Therefore these are also captured, thereby subsuming the DMT policy. This is in addition to the architectural improvements in Polyflow over DMT architecture.

Figure 8.1 shows the performance of this policy on a 4-core system when compared to that of a single core of the system. The configuration is described in section 5.5.1. Note that the closest postdominator policy performs much better than the reported performance for either the Skipper or DMT architecture. Compared to Skipper, it captures a much wider set of task opportunities than just the immediate postdominators of low-confidence branches. Further, the evaluation system is a 4-core speculative parallelization system, whereas the tasks spawned in Skipper shared the fetch and execution bandwidth of a single superscalar core. Compared to DMT policy, this policy captures a much wider set of tasks that includes tasks that jump over hammocks and that spawn within inner loops.

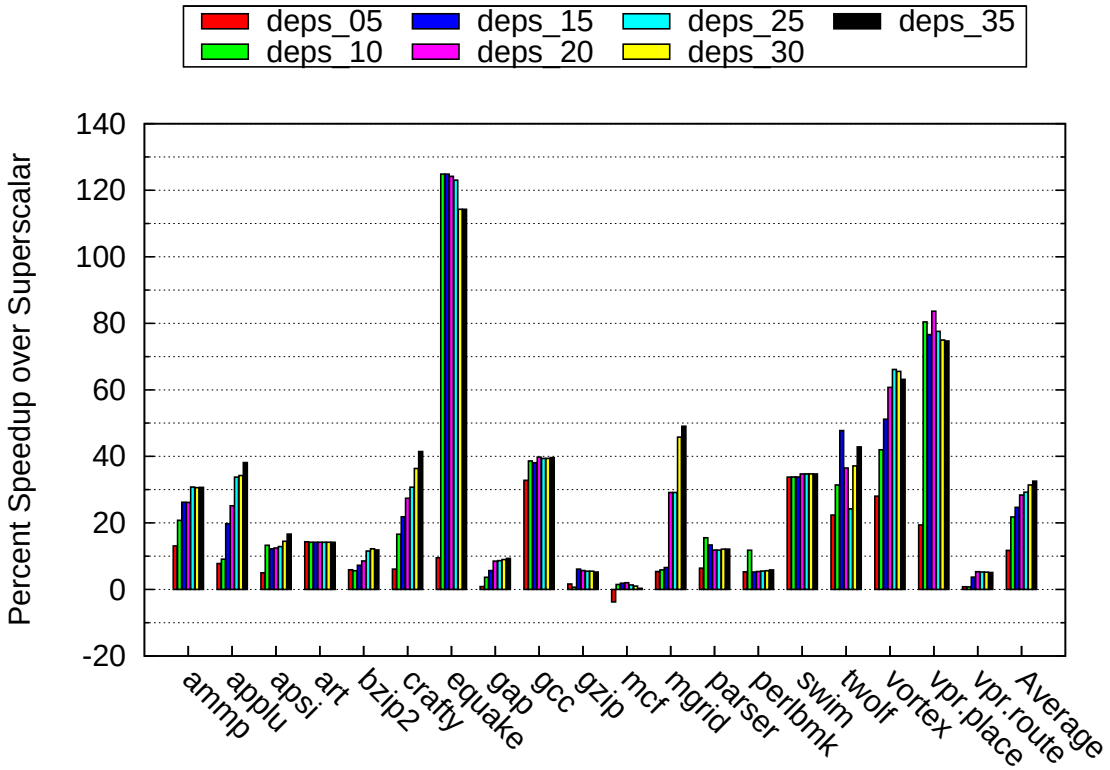

Figure 8.2: Performance of a speculative parallelization system where task selection is an approximation of the Multiscalar policy. The selection policy places a maximum threshold on number of data-dependences allowed to cross task boundary to eliminate potential tasks. The graph shows performance for different thresholds.

8.1.2 Data-Dependence Count Policy

The next comparison policy used in this study tries to capture the insight behind Multiscalar policy [68]. The policy starts with postdominator analysis to get potential spawner-spawnee pairs. The spawnee pairs are control-independent of the spawner points, thereby eliminating task squashes from control misspeculation (Control Flow Heuristic). Next, tasks that are too large (larger than 1K instructions on average) or too small (less than 10 instructions) are pruned out (Task Size Heuristic). Finally, a limit is placed on the number of data-dependences that can cross the task boundary. Tasks choices that cause too many data-dependences to cross task boundary as determined by a threshold are eliminated (Data Depenence Heuristic).

Figure 8.2 plots the performance of this selection policy, for varying thresholds on the crossing data-dependences. Note that the dependence count uses profile information as opposed to just counting using a static dependence graph. Therefore it is more accurate in that it accounts only for the data-dependences that are exercised at run-time.

There are several points to note here. There are some benchmarks where imposing a strict

threshold (i.e. fewer dependences) is sometimes better than a more relaxed threshold. For example, for equake, allowing 30 or more inter-task dependences degrades the task selection. Similarly, in twolf, moving beyond 15 dependences makes the performance worse. But overall, it is profitable to not restrict the count of inter-task data-dependences.

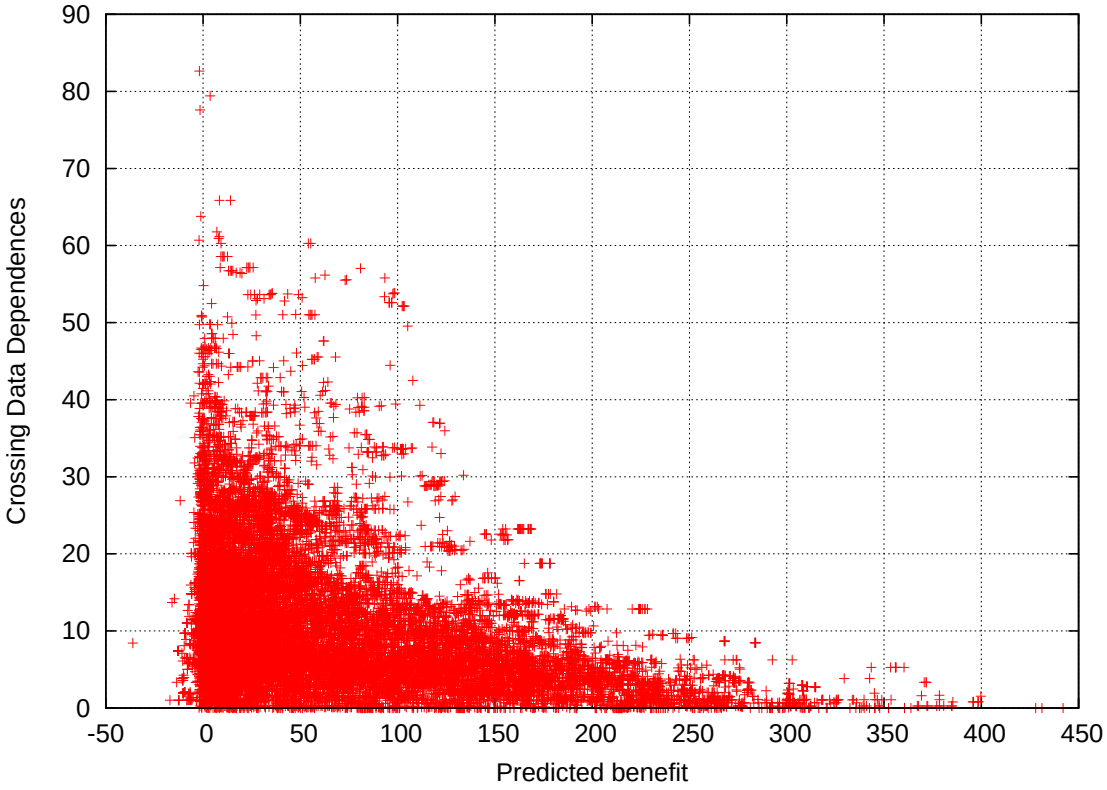

Figure 8.3: Relationship between the number of crossing data-dependences and the predicted benefit of a task using the model of chapter 5 and for the benchmark gcc.

There are several reasons why this heuristic doesn't work in the Polyflow system. As mentioned above, Polyflow tries to synchronize on frequent data-dependences, and also implements a non-blocking scheduler. Therefore, the number of inter-task data-dependences is not a good metric of the benefit of a task. Rather, it is the minimum of the adjusted slack on all such inter-task data-dependences for a given task, as explained in section 5.3. Even a single dependence can make a task perform poorly if it has low slack. On the other hand, a task with a large number of data-dependences crossing the task boundary might still be profitable if all the dependences have ample slack.

Figure 8.3 plots the relationship between the number of data-dependences crossing the boundary for each potential task in gcc and the average benefit of spawning that task alone as predicted by the model of chapter 5. The model was validated to be accurate (figure 5.12). The figure shows that there is little correlation between the number of data-dependences and task benefit. In particular, several tasks with few data-dependences perform quite poorly. On the other hand, tasks

with a relatively large number of inter-task data-dependences (e.g. between 50 and 60) sometimes perform reasonably well.

8.2 Task Selection in Polyflow

Task selection in Polyflow involves two stages. First the model of chapter 8 is used to place a threshold on individual task behavior. The next stage incorporates containment relationship between tasks to improve on the task selection.

8.2.1 Impact of Threshold

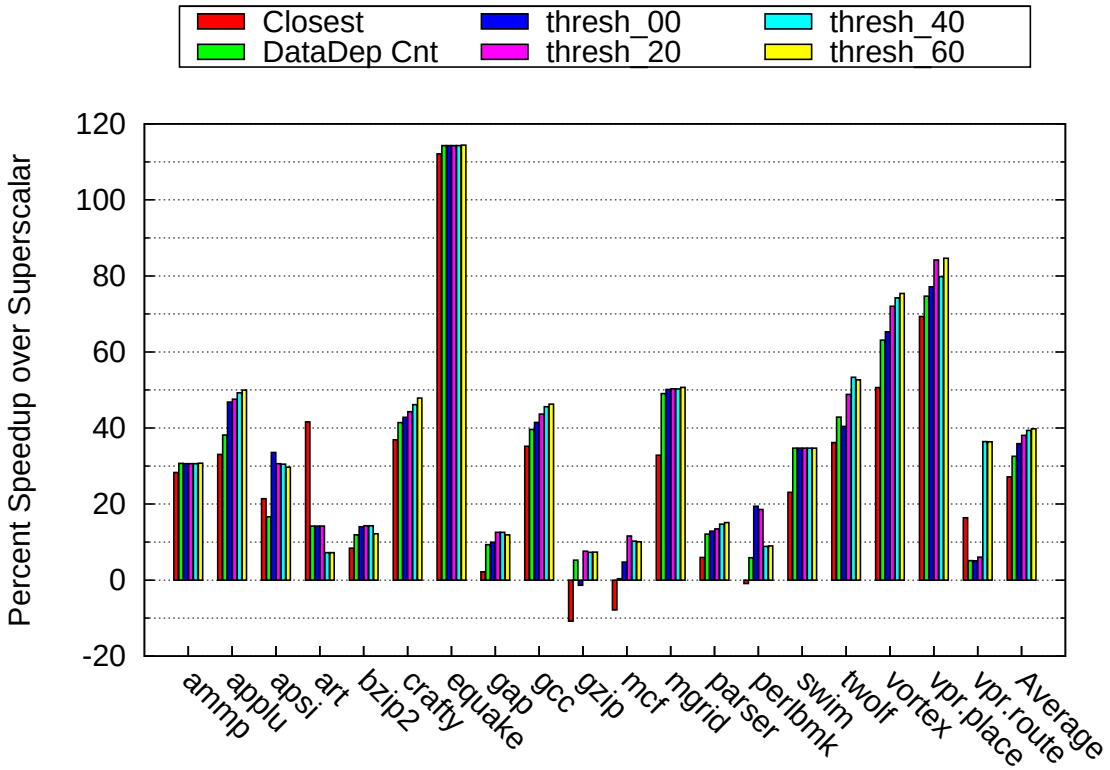

Figure 8.4: Impact of placing a threshold on task benefit as predicted by the model of section 5. Performance is shown as speedup of a 4-core Polyflow system over a single superscalar core. Closest and Data-dependence Count (with count of 35) spawn policies are shown for comparison.

The first step in making a task selection involves placing an admittance criterion on individual task performance. This uses the predictions from the task benefit model to estimate individual task behavior. Figure 8.4 shows the effect of placing a per-task threshold on performance. The Closest spawn policy of section 8.1.1 and the Data-dependence Count policy of section 8.1.2 are shown for comparison.

There are several points to note. Firstly, just placing a threshold of 0 can lead to dramatically improved performance, by pruning out task choices that degrade performance. Mcf and gzip illustrate this case, where most of the tasks degrade performance and these are eliminated. However, the impact of placing stricter thresholds can vary depending on the utilization and contention for thread resources. Benchmarks like twolf and vortex have a large number of task choices that contend for limited cores and therefore stricter threshold work better in these benchmarks. On the other hand, for cases like perlbmk, too strict thresholds leave few task choices and leads to lower performance. A simple solution is to select a medium threshold (e.g. 20 or 40) which would work reasonably well for most benchmarks.

An interesting case is observed for art, where the closest spawn policy outperforms both Data-dependence Count and threshold-based spawn policies. This is because of the minimum size of 10 used in both these policies. This improves performance for overall except for art. Art spends a lot of its time in a very small but long-running loop which creates a difficult choice for the system. It can either spawn very small tasks for each inner-loop iteration, or not spawn at all since the loop fall-through is quite distant. The minimum task size of 10 causes this loop to become Almdahl's sequentializing bottleneck in threshold-based policy. Such loops could perhaps be tackled with transformations like unrolling or strip-mining.

8.2.2 Nesting Analysis for In-order Task Spawning

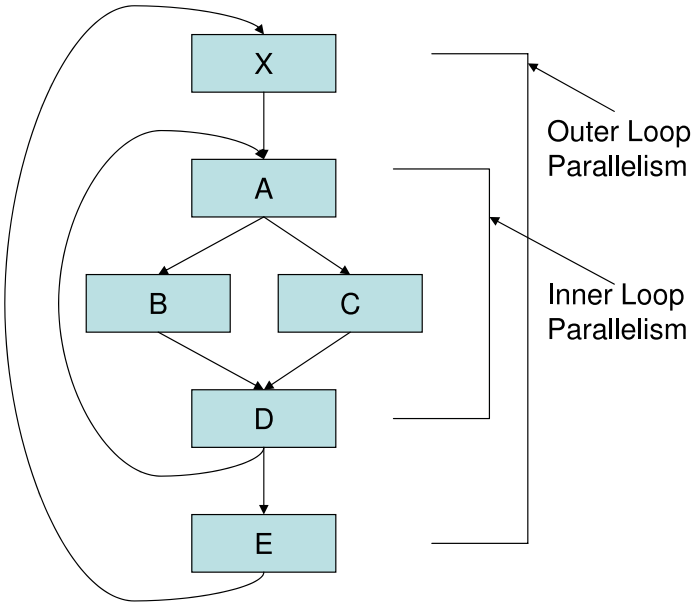

Figure 8.5: An example of nested loop. Polyflow has to decide between exploiting inner loop parallelism and parallelism in the outer loop.

The above policies as well as the task selection policies in most speculative parallelization systems

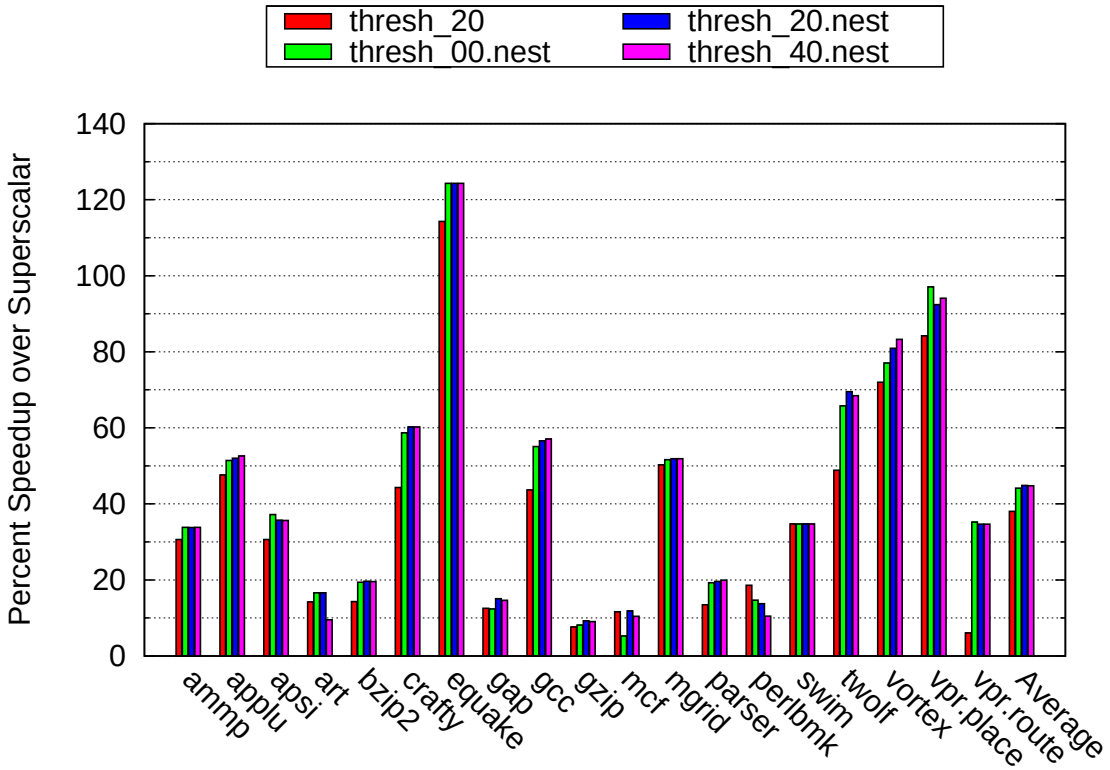

Figure 8.6: Performance impact of incorporating nesting relationships between tasks along with a threshold on individual tasks.

consider task behavior in isolation. However, most speculative parallelization systems spawn tasks in-order which means that once a task has spawned a later task, all the spawn opportunities that lie between these two tasks will be ignored (unless the later task is squashed for some reason). This restriction is particularly important for selected tasks that are nested within other selected tasks, or overlap partially with other tasks. Such tasks are unlikely to be spawned even though they were selected. But often nested tasks might perform better than the task that encloses them. For example, for nested loops, spawning within the inner loop might be better than spawning in the outer loop. This might be influenced by a number of factors including data-dependences. In other cases, however, the enclosing task might be better. Thus, it is important to spawn at the correct nesting level. Figure 8.5 illustrates the choice facing Polyflow for the case of a nested loop.

In Polyflow this problem is approached by constructing a nesting graph of all potential tasks that pass an individual benefit threshold. This is a generalization of the loop graph construct [62]. It is a directed graph. Each node in the graph represents a potential task. Edges are directed and represent containment relationship. Thus, if a task B is contained in a task A such that there is no other task that completely contains B and is contained in A, then there is an edge from A to B. The task B might itself contain other smaller tasks, so there might be an edge from B to other tasks.

At each node, information is stored about the task being represented. The information stored is the

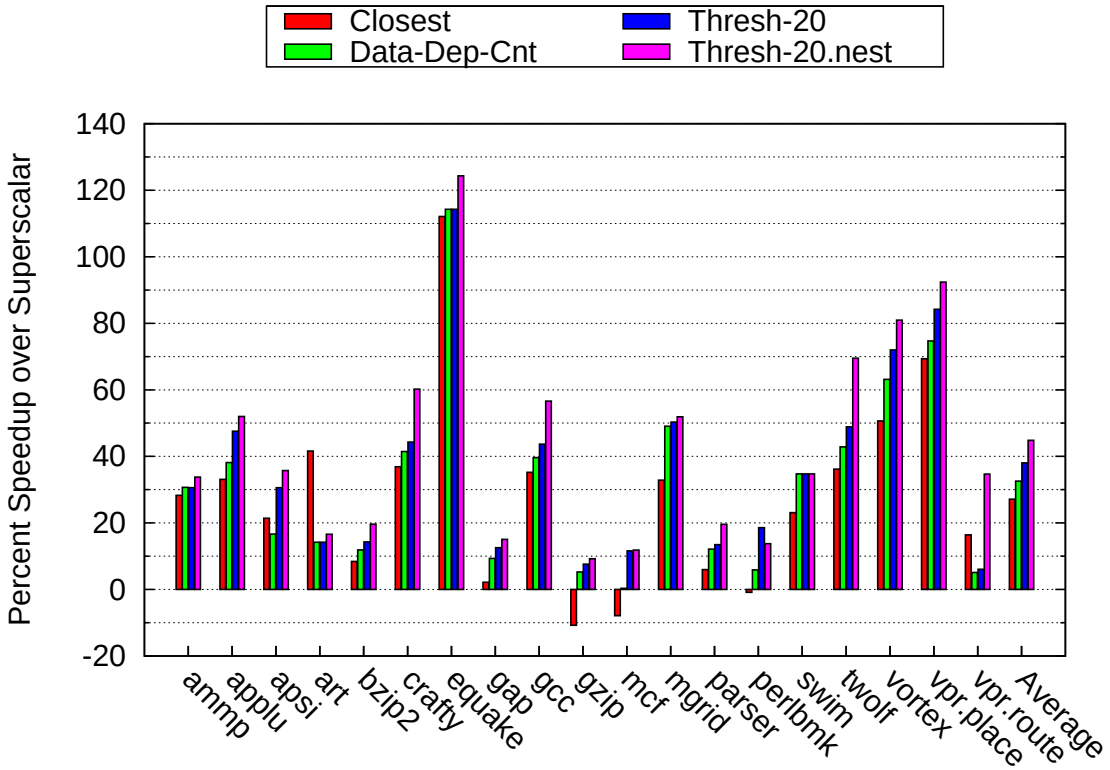

Figure 8.7: A summary of performance achieved by different task selection policies.

count of the task for the containment represented as well as the average benefit of the task. Note that a task B might not be contained within another task A for all of its occurrences, therefore only the contained instances of B are tracked for the path A→B. This graph can be used to find cases where an overall better task is nested inside an inferior choice (the metric is contained count times average improvement per instance). All successors are considered, not just the immediate successor. Therefore, the graph is traversed all the way down to leaf nodes. If a superior nested task is found, the inferior outer spawn choice(s) can be eliminated.

The policy implemented here also tries to incorporate the effect of limited cores. Sometimes a nested choice can be more profitable but might need to spawn many more instances. In such a case, the outer task might be better because it uses cores more judiciously. The solution used here is to disable the outer task if an inner task delivers higher performance without requiring more than twice the number of cores.

Figure 8.6 shows the result of incorporating nesting analysis into task selection. This can lead to significant improvement in performance for several benchmarks. The notable gainers are crafty, gcc, twolf, and vpr.route. The default behavior, without nesting analysis, is to spawn the first spawn choice that becomes available. This gives preference to outer tasks over the tasks that they enclose. Nesting analysis can disable such enclosing tasks when better choices exist inside. For the

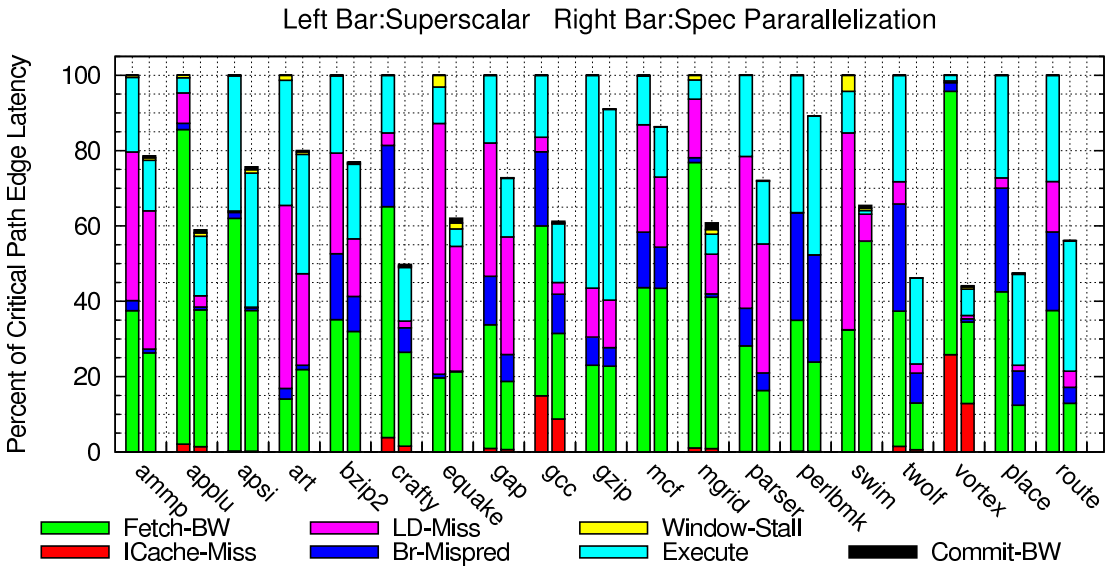

Figure 8.8: Critical path edge latency breakdown for superscalar and speculatively parallel executions.

above-mentioned benchmarks, there are several nesting levels comprising procedure calls, loop and hammock branches. Spawning at the right nesting level is therefore important to performance. Figure 8.7 deconstructs the gains achieved by the Polyflow task selection policy. Overall it represents an average of 12% improvement in performance over the Multiscalar heuristic, which comes from having a better model and from nesting analysis. The nesting analysis itself is enabled by the accurate model of chapter 5, since it needs to compare the behavior of different potential tasks.

8.3 Understanding Performance

Section 8.2 shows impressive performance for Polyflow task selection policy. This section tries to analyze the source of those performance gains. Figure 8.8 shows the breakdown of critical path latency for a set of benchmarks, for superscalar execution as well as for Polyflow execution. The Polyflow selection policy attacks a variety of application performance bottlenecks, and thus its gains come from a wide range of sources:

- **Fetch Benefit**: Limited fetch (and execution) bandwidth in a single thread unit restricts the peak fetch rate, even though additional parallelism exists in the region. This corresponds to the traditional notion of “parallelism”, that is, independent instructions that could not be fetched/executed earlier due to insufficient bandwidth. Spawning allows additional fetch units to be used, allowing for a higher peak fetch rate. E.g. apsi and vortex. Corresponds to “Fetch-BW” bar.

- **Instruction Cache Benefit:** Instruction cache misses stall the frontend and delay the fetch of future independent instructions even if they are already present in the cache. Spawning allows multiple units to fetch instructions independently, misses in one thread need not stall fetch in another unit. E.g. gcc and vortex. Corresponds to “Icache-Miss” bar in the graph.
- **Branch Misprediction Benefit:** Mispredicted branches delays fetch of all future instructions, even those that are control-independent of the branch. Spawning in control-independent regions of the program allows other threads to keep making progress, even when one thread gets stuck on a mispredicted branch. E.g. crafty, twolf, vpr.place and vpr.route. Corresponds to “Branch-Mispred” bar.
- **Window Benefit:** Long latency load misses and limited parallelism in current region stall the frontend due to a full reorder buffer or scheduler until space frees up. Spawning creates a larger (distributed) window of instructions for exploiting parallelism further out, or for getting to data misses sooner. Resource stalls in one unit need not stall fetch in another unit that has space in its buffers. E.g. equake and swim. This comes from a combination of the ”Window-Stall”, ”LD-Miss” and ”Execute” bars.

Note that some of the bars might actually increase rather than become smaller. For example, in swim, the “Fetch-BW” contribution to critical path becomes larger in Polyflow execution. The original critical path was dominated by load-misses and the window stalls induced by these misses. Polyflow was successfully able to remove a large number of these misses and stalls from the critical path. The next longest path became the new critical path and comprised of many fetch edges in this case. This indicates that there is even more parallelism in Swim which the current architecture was unable to exploit due to limited cores and from the restriction of spawning tasks in-order. On the other hand, increases in contribution of “Execute” component (e.g. Vortex) implies that the execution is starting to approach dataflow height and that limited unexploited parallelism might exist on this architecture.

PART III

ENHANCING PARALLELISM THROUGH BOTTLENECK REMOVAL

CHAPTER 9

APPLICATION BOTTLENECKS TO PARALLELIZATION

This chapter addresses the challenge of performance debugging the parallelization process. For some applications, implicit parallelization approaches work well “out of the box” and sufficient performance is attained. But for many applications, the first-cut effort leads to low improvements and a feedback mechanism is needed to help iteratively improve the application until sufficient performance is attained.

A major challenge in parallelization of applications comes from inter-thread or inter-task data-dependences that pose bottlenecks to parallel performance often in unforeseen (and usually unintended) ways. Parallelization, whether implicit or explicit, needs performance debugging to weed out such bottlenecks. However, such performance debugging requires time and effort. A lack of tools to aid in this step makes parallelizing single-threaded applications a very time-consuming and ad-hoc process.

This chapter develops a tool to analyze single-threaded applications for parallelism. The tool is called “**S**oftware **Par**allelization Bo**t**tleneck **An**alyzer” or SPARTAN. SPARTAN identifies data-dependences in applications that are likely to constrain parallelization, and therefore pose bottlenecks to parallel performance. The tool also quantifies the estimated performance benefit of alleviating a particular bottleneck (or of specified combinations).

9.1 Background

SPARTAN is built around two key concepts: first, it does a dependence height study to estimate the potential of parallelizing an application(section 9.1.1); and second it does a critical path analysis [3] to identify the bottlenecks posed by data-dependences in parallelizing the application.

9.1.1 Abstract Dependence Height Analysis

An abstract dependence height study is a useful tool to get an upper bound on the performance potential of parallelizing the application. One such study was carried out by Lam and Wilson [7], and described in section 3.1. The SP-CD-MF configuration in their study can be used to find the dependence height of the program, which is the best any parallelization of the program could achieve. This is because control-independent flows in the study represent a superset of the tasks that could be simultaneously executing instructions. On the other hand, the above study is hugely

optimistic in many ways, including allowing for an infinite fetch bandwidth to each flow, and the ability to buffer an infinite number of instructions in each flow. This is something that cannot be achieved by a real task unit.

SPARTAN's dependence height study, on the other hand, imposes constraints on individual flows corresponding to the constraints imposed on a real task unit (limited fetch bandwidth, buffer constraints, etc). But it still allows for infinitely many flows to be executing simultaneously. This is because limiting the number of flows would require making an optimal resource allocation decision to calculate the upside potential, which is a hard problem. In addition, the study doesn't impose any cost to flow creation or inter-flow data communication. Thus, it obtains the benefits of spawning each flow without incurring the costs seen in a real system. Again, finding the upside potential that includes these costs would require making an optimal flow selection to balance the benefits and costs of flow creation, which is a hard problem. Note also that Lam and Wilson's study did not enforce reassociatable data-dependences such as those due to loop index variables. SPARTAN does enforce these dependences and identifies them as bottlenecks if they have an impact on performance.

In spite of the idealizations that remain, this version of the dependence height study is a more useful tool to understand the amount and nature of parallelism in the application, and the bottlenecks to further parallel performance. The next sections describe how to identify these bottlenecks. Note that the rest of this chapter will not differentiate between a task and a flow for easier reading.

9.1.2 Critical Path Analysis

In order to identify the bottlenecks to parallelism, SPARTAN does a critical path analysis for the dependence height study described above. A dependence graph is constructed for the dependence height study. This graph is based upon Fields' dependence graph of superscalar execution [3] described in section 3.2.1. Section 4.3.2 described how that model can be extended to capture control-independent task spawns required for the above study. No extra costs are associated with task-related edges. The longest path from the first fetch node in the dependence graph to the last commit node provides the critical path that determines performance after parallelization. Dependence edges on the critical path can provide useful information about the bottlenecks to parallel performance.

9.2 Design of SPARTAN

SPARTAN is a software tool built on top of a trace generator. It analyzes a run of a single-threaded application and lists data-dependences that will pose performance bottlenecks when the application is parallelized. Internally the tool performs a dependence height analysis on the application trace. This allows it to identify limiting data-dependences independently for any particular task partitioning of the application. Next, SPARTAN breaks down the program critical path to isolate

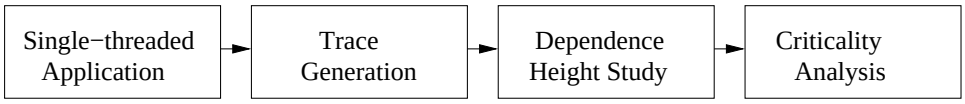

Figure 9.1: Bottleneck identification steps.

top bottleneck data-dependences in the program. SPARTAN can also estimate the improvement in parallel performance potential from removing a particular bottleneck dependence (or a combination of dependences).

9.2.1 Functionality

SPARTAN works in two modes: bottleneck identification mode, and bottleneck quantification mode. In the identification mode, the tool takes in a single-threaded application, and outputs a list of data-dependences that it believes will pose important bottlenecks to parallel performance, ranked by their relative importance. In the bottleneck quantification mode, it takes in, along with the single-threaded application, a data-dependence (most likely identified in the identification phase) or a list of such dependences, and outputs the expected improvement in parallel performance potential when the bottleneck posed by that dependence is removed. Thus repeated queries to SPARTAN can help identify the order in which bottlenecks should be removed.

9.2.2 Bottleneck Identification

SPARTAN is a software tool built on top of a trace generator. Figure 9.1 illustrates the steps involved in bottleneck identification process. The trace generation step outputs the trace for the single-threaded application running on a given input. This step can use a trace generator similar to Intel's PIN [79]. This trace is passed on to the dependence height analysis step, which assigns a timestamp to each instruction for its fetch (F), execution (E) and commit (C), following the techniques of trace-based simulation of Wall [6] and modeling the constraints of the dependence height study described in section 9.1.1. As described earlier, the dependence height study assigns timestamps to each instruction to estimate the potential of the best-case parallelization of the application in its current form. The next steps analyze chunks of trace at a time for good performance.

These timestamps can then be used in the construction of the program dependence graph for the dependence height study. This construction is done using the concepts of section 3.2. Once the dependence graph has been constructed, SPARTAN computes the longest path from the *start node* of the graph to its *end node*. This path represents the *program critical path*. Of particular interest are the data-dependences on the critical path that cross flow boundaries. These represent a superset of dependences that could pose a bottleneck to parallel performance, since the flows represent a superset of tasks that could be created. The tool records all such dependences. In the end,

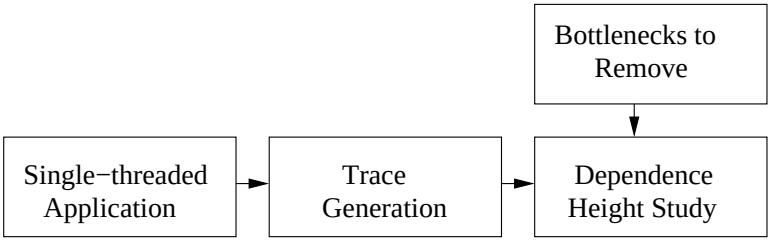

Figure 9.2: Bottleneck quantification steps.

SPARTAN outputs the list of such observed critical data-dependences, sorted by the number of times a particular dependence appears on the critical path through the trace.
Note that SPARTAN's data-dependence list is not tied to any particular selection of tasks to parallelize the application. This makes it particularly suited for refactoring a single-threaded application for which an appropriate task partitioning is not yet known. SPARTAN could be modified to identify the bottleneck dependences for a specific choice of tasks. The timestamp assignment for a given task selection could be done using the dependence graph concepts of section 4.3.2. Note also that instruction criticality analysis is typically used to identify architectural bottlenecks, such as cache ports, branch prediction, etc. SPARTAN is a novel use of the critical path analysis to learn about the application structure and data-dependences.

9.2.3 Bottleneck Quantification

The bottleneck quantification mode allows SPARTAN to estimate improvement in parallel performance potential from removing a particular bottleneck dependence (or a combination of dependences). The steps involved are illustrated in figure 9.2. The trace generation step proceeds as before. In this mode, however, the programmer provides a list of the data-dependences that are to be ignored. This information is incorporated in the dependence height analysis, where the specified dependence constraints are not enforced. That is, the consumers of these dependences are allowed to execute even before the specified producers have completed execution (other dependences still need to be satisfied for the consumers). The result of this study gives the upside potential of parallelization when the specified bottleneck is removed.

9.3 Bottleneck Analysis for Benchmarks

This section describes the results of running SPARTAN on a set of single-threaded benchmark applications from the SPEC2000 suite, and gives some insights about the type of bottlenecks observed. For the purpose of this study, the following SPEC Integer applications were analyzed: vpr.place, twolf, parser and gzip. They are all single-threaded C applications. Table 9.1 gives brief descriptions. These benchmarks are run on the Minnesota reduced input sets [45] for a representative interval of 100M instructions. The next section presents the output from running

Benchmark	Description
175.vpr	FPGA Circuit Placement and Routing
300.twolf	Place and Route Simulator
197.parser	Word Processing
164.gzip	Compression

Table 9.1: SPEC Benchmarks chosen

```
int my_irand(int imax) {
  /* Create random integer between 0 and imax */
  int ival;
  current_random = current_random * IA + IC;
  ival = current_random & (IM-1); /* Modulus */
  ival = (int)((float)ival * (float)(imax+0.999) /
              (float)IM);
  return (ival);
}
```

Figure 9.3: VPR Place Random number generator with bottleneck dependence highlighted.

SPARTAN on these applications, along with descriptions and code snippets to illustrate the nature of bottlenecks.

9.3.1 Bottlenecks in VPR Place

PC→PC	From	To	Count
2688→263c	my_irand	my_irand	32611
9604→9604	try_swap	try_swap	26129
2710→263c	my_frand	my_irand	25048
9604→967c	try_swap	try_swap	13360
26bc→92cc	my_irand	try_swap	12729
d39c→95fc	net_cost	try_swap	11679
92cc→9330	try_swap	try_swap	11272
2688→26d4	my_irand	my_frand	10775

Table 9.2: Bottleneck dependences in VPR Place sorted by the count of the number of times a particular dependence appeared on the critical path.

Table 9.2 lists the output of running SPARTAN on VPR Place. The tool identifies the address of the producer and consumer instructions of the critical dependences. The tool annotates this information with the function names of these instructions. This information can also be used to trace back the lines in the code causing this dependence.

For VPR Place, quite surprisingly, a major hindrance to parallelization comes from the dependence between successive calls to random number generator functions: `my_irand` and `my_frand`, the integer and floating point versions. These account for 3 of the top 8 bottlenecks. Figure 9.3 illustrates the code that causes this problem. The variable `current_random` is first read, and

```
new_dbox(antrmptr, costptr)
int *costptr;
{
  for (termptr=atnrmptr;
       termptr;
       termptr=termptr->nextterm){
    ...
    for (netptr=dimptr->netptr;
         netptr;
         netptr=netptr->nterm){
      oldx = netptr->xpos;
      if (netptr->flag == 1) {
        ...
      } else {
        ...
      }
      *costptr += ABS(newx-new_mean) -
                  ABS(oldx-old_mean);
    }
  }
  return;
}
```

Figure 9.4: `new_dbox` function in Twolf highlighting the bottleneck integer reduction.

then written in each call to `my_irand` (and also in `my_frand`). This forces a sequentialization of successive executions of the function, which becomes a bottleneck to parallel performance.

9.3.2 Bottlenecks in Twolf

PC→PC	From	To	Count
1cfa4→1cf54	Yacm_random	Yacm_random	70238
aaf8→ ab0c	new_dbox	new_dbox	28795
9dc8→ 9ddc	new_dbox_a	new_dbox_a	23161
1cfa4→1cfc4	Yacm_random	Yacm_random	8031
ab10→ aaf8	new_dbox	new_dbox	7687
9de0→ 9dc8	new_dbox_a	new_dbox_a	5908

Table 9.3: Bottleneck dependences in Twolf sorted by the count of the number of times a particular dependence appeared on the critical path.

The story in Twolf is quite similar to that of VPR Place. Table 9.3 lists the top bottlenecks identified by SPARTAN. As with VPR Place, random number generation is a major bottleneck to parallelization. In addition, this study investigated the bottleneck in the functions `new_dbox` and `new_dbox_a`. These functions are nearly identical, as is the bottleneck, which comes from an integer reduction illustrated in figure 9.4. The variable `costptr` is an integer location that is read and written in each iteration of a doubly nested loop. This prevents parallelization of both the inner and the outer loop. This variable can easily be reassociated with large improvements in the

potential.

9.3.3 Bottlenecks in Parser

From	To	Count
xfree	xfree	462577
xfree	xfree	21548
build_clause	build_clause	7776
free_disjuncts	free_disjuncts	5652
right_table_search	right_table_search	4322
power_prune	power_prune	4223

Table 9.4: Bottleneck dependences in Parser sorted by the count of the number of times a particular dependence appeared on the critical path.

Parser highlights another type of bottleneck dependence, arising from the allocation and freeing up of memory in programs. This benchmark has its own memory management functions: `xalloc` and `xfree`, and the top two dependences (where the third bottleneck is very distant) come from the dependence within the function `xfree`. Again, there is a dependence between successive calls to `xfree` which prevents any meaningful parallelization of the application. But the dependence could be easily removed by calling parallelizable allocators explored in literature.

9.3.4 Bottlenecks in Gzip

PC→PC	From	To	Count
b3f8→b400	updcrc	updcrc	766811
b3ec→b3f4	updcrc	updcrc	766811
b5d8→b5e0	flush_window	flush_window	425562
b5cc→b5d4	flush_window	flush_window	425553
10ec→110c	deflate	deflate	216305
10fc→1110	deflate	deflate	215441
1110→1118	deflate	deflate	215305
110c→10c8	deflate	deflate	209197

Table 9.5: Bottleneck dependences in Gzip sorted by the count of the number of times a particular dependence appeared on the critical path.

Finally, a different class of dependences that limit parallelization was found in Gzip, listed in table 9.5. These dependences have more to do with the choice of the algorithm which is inherently sequential in nature, and less with the choice of library functions or implementation details. This is illustrated through the `updcrc` function in figure 9.5. The `crc` variable introduces sequentialization between successive calls to `updcrc` and parallelizing this requires domain expertise and knowledge of the algorithm used.

```
/* Run a set of bytes through the crc
   shift register. */
ulg updcrc(s, n) {
  /* temporary variable */
  register ulg c;
  /* shift reg contents */
  static ulg crc = (ulg)0xffffffffL;
  if (s == NULL) {
    c = 0xffffffffL;
  } else {
    c = crc;
    if (n) do {
      c = crc_32_tab[((int)c ^ (*s++)) & 0xff] ^
          (c >> 8);
    } while (--n);
  }
  crc = c;
  return c ^ 0xffffffffL;
}
```

Figure 9.5: `updcrc` function in Gzip highlighting the sequential nature of algorithm.

9.3.5 Discussion

The observed bottleneck dependences can broadly be classified into two categories:
Essential dependences: These arise from the choice of high-level algorithm used in that it is inherently sequential in nature, such as the case of Gzip. Parallelization would require domain expertise. SPARTAN can be used as an aid to figure out the algorithms to redesign.
Accidental dependences: These are related to the specific way in which an otherwise parallelizable task was coded up, therefore constraining parallelization. Examples are random number generation in VPR and Twolf, memory allocation in parser, etc.
The latter case is the more interesting one, and is surprisingly frequent in the above case studies. The fact that this is observed in SPEC Integer benchmarks, that are traditionally believed to not be amenable to parallelization, makes it even more impressive. Encouragingly, the trend of accidental dependences from within library functions shows a clear path forward, in the form of parallel library versions that can be widely used to ease the task of the parallel programmer.

9.4 Quantifying Bottlenecks and Validation

This section does a case study of the VPR Place application, and validates that the predictions of SPARTAN are relevant for an actual parallelization of the application.

9.4.1 Quantifying Bottlenecks in VPR

This study runs SPARTAN in the bottleneck quantification mode for VPR and gets the results shown in table 9.6. The potential is shown as times speedup (X) over the performance (measured

in instructions per cycle or IPC) on a single aggressive 4-wide out-of-order superscalar processor. There is almost a 20-fold improvement in the performance potential, from 8.3X to 154.2X over superscalar performance. This indicates that removing this dependence perhaps through parallelizable or parallelized random-number generation could lead to huge rewards.

Upside potential in original form	Upside potential with bottleneck removed
8.3X	154.2X

Table 9.6: Impact of removing the bottleneck from random-number generation on upside potential of parallelization in VPR Place.

9.4.2 Potential for Parallel Performance on Polyflow

The next step is to test the applicability of results from SPARTAN on an actual parallelized version of VPR Place. This step is carried out by implicitly parallelizing VPR Place on a Polyflow system. Since Polyflow needs to buffer processor state until it can be committed to the system, this limits the scope of parallelization. In particular, the system cannot create arbitrarily large tasks, since these would exceed the allowed buffer. The system therefore allows tasks that have been profiled to have an average length of at the most 1K instructions.

Upside potential in original form	Upside potential with bottleneck removed
7.0X	14.9X

Table 9.7: On the evaluation system, impact of removing the bottleneck from random-number generation on potential of parallelization .

This reduces the upside potential of application parallelization. Another dependence height study is carried out to estimate the new upside potential of parallelization under this constraint (table 9.7). The upside potential is found to drop from 8.3X to 7X when task size is restricted to capture the above described constraint. In addition, the impact of removing the bottleneck from random number generation is also considerably lower, from almost 20-folds down to 2-folds improvement. This points to the fact that removing the bottleneck especially benefits large tasks where earlier they were of little use (due to this dependence). This result also indicates a bottleneck from the architecture (buffer size) as was also identified previously. Nevertheless a 2x improvement in performance potential is still worth pursuing and the next section converts this potential into useful performance.

9.4.3 Speculative Parallelization of VPR

Having obtained an idea of the upside potential of implicit parallelization from SPARTAN from removing the bottleneck, this section carries out an actual parallelization on the evaluation multi-core system. First the application is partitioned into tasks as described in section 8.2. The output of the task selection phase specifies how to break the application up into tasks. The Polyflow system then internally parallelizes the application according to that task selection, while giving the appearance of sequential execution externally. This study doesn't constrain the number of cores available in the system to allow a maximal exploitation of parallelism. Note that other costs and constraints are still imposed as described in chapter 6.
The following improvements in performance are achieved on the evaluation Polyflow system:

Performance Improvement in original form	**Performance Improvement with bottleneck removed**
2.3X	4.5X

Table 9.8: Measured performance on an implicit parallelization system, before and after removing the bottleneck in VPR Place.

Note that when the bottleneck is removed, a new task selection optimized for the new version of the application is generated. The achieved performance is significantly lower than the predicted upside potential. This is because the experimental system incurs significant cost from inter-task data synchronization and other task-related actions. Nevertheless, the achieved gains are quite encouraging, and almost a two-folds increase in parallel performance is observed from removing the bottleneck. This improvement is quite similar to the gains predicted by SPARTAN.
This improvement was further explored by identifying the top bottleneck dependences for implicit parallelization of VPR in its original form. The top bottleneck dependence for this parallelization was found to be same as identified by SPARTAN through its abstract dependence height analysis, and listed in table 9.2. In fact, the top dependence lists match up quite well. This is indicative of two things. Firstly, the Polyflow task selection phase generates a selection that is optimized for the underlying architecture, and thus gets quite close to the application bottlenecks. Second, SPARTAN through an abstract dependence height analysis is able to make quite meaningful and useful predictions about the behavior of the application when parallelized on a real system.

CHAPTER 10

ARCHITECTURAL BOTTLENECKS TO PARALLELIZATION

Limit studies such as the dependence height study of Lam et. al [7] indicate high upside performance potential for speculative parallelization. In practice, however, most speculative parallelization systems achieve much lower performance than the upside potential predicted by such limit studies. A major reason for this is that upside potential studies capture all possible benefits of speculative parallelization without imposing many of the costs and constraints to parallelization that arise in real systems. This chapter revisits major architectural constraints and costs, and tries to quantify the impact of each of those factors on the performance achieved.

10.1 An Upside Potential Study

This section tries to quantify the upside potential of speculative parallelization. The study is carried out for the application in its original form without any compiler transformations for parallelization. There can be several transformations such as those described in section 7.4 that improve the parallelizability of an application. However, incorporating the impact of parallelizing transformations in the upside potential study is a computationally intractable problem because it will require exploring, for each region of the application, all possible combinations of transformations applicable. This study therefore limits itself to the application in its current form. Chapter 9 tries to relax this restriction to enhance application parallelizability.

10.1.1 Methodology

This study builds upon the dependence height study of Lam et. al [7] that was previously described in section 3.1. Lam's study was interested in the upside potential of parallelization based on just the application behavior and without considering the impact of any architectural constraints. The objective of the current study is to incorporate constraints that are likely to be encountered in a speculative parallelization architecture. But at the same time, the study idealizes some of the costs and resource constraints imposed by current architectures to leave room for better mechanisms and resource allocation policies in future architectures.

The high-level methodology is similar to Wall's trace-analysis technique [6]. The study analyzes an application trace, and tries to compute the earliest time at which the trace could have completed, based on the application and architectural constraints modeled. Table 10.1 summarizes the details

Constraint	Details
Application Dependences	True data-dependences, control-dependences that can't be predicted correctly
Architectural Dependences Within a Task	All constraints enforced within superscalar execution, in-order fetch and commit, buffer constraints
Inter-Task Constraints	No extra penalty for inter-task data-dependences, task spawn or task reconnect actions. No limitations on number of simultaneous tasks.
Latency on Edges	EE, EC, EF edges have same latency as in superscalar execution. FF edge latency within task models limited fetch bandwidth. CC edges have zero latency. Inter-task spawn FF edges have zero latency.

Table 10.1: Summary of Upside Potential Study

of the upside potential study.

The study starts with the SP-CD-MF configuration in Lam's study. This is a very aggressive configuration that enforces only the control-dependences that cannot be speculated upon successfully. In addition, it allows for multiple flows, so an instruction can be executed as soon as its control-dependences (that were not correctly speculated upon) and its incoming data-dependences are resolved.

This study will use the term task instead of flow to avoid any confusion. For the current study, task spawn points are obtained through a compiler postdominator analysis. Spawner points are the ending points of basic blocks, which are usually branches and other control-transfer instructions. In addition, for loops consisting of a single basic block, additional spawn points are included. These points are chosen so as to avoid making the dependence on induction variables cross task boundary. The spawnee point for any spawner is the nearest postdominator of the spawner point. Since the other postdominators of the spawner task are also postdominators of the spawned task, these would be spawned at some point in the future by a later task. Note that tasks can be spawned in a nested manner, because new tasks might become available as control dependences are resolved.

10.1.2 Architectural Constraints Modeled

In addition to application control and data dependences, this study models other dependences to capture architectural constraints. This is best visualized in terms of a dependence graph. Lam's study had a single node for each instruction, representing its execution. This study models the Fetch, Execute, and Commit of each instruction as separate nodes, similar to Fields' dependence graph model [3]. This is done by keeping separate timestamps for each instructions's Fetch, Execute and Commit. Thus application data-dependences are modeled as EE edges and control-dependences as EF edges.

Other constraints in a speculative parallelization (SP) architecture are also modeled. Since SP architectures preserve sequential semantics, instructions are committed in-order, which is captured through CC edges between successive instructions. Lam et. al associated a unit latency with each

edge. This study associates the execution latency imposed by function units in a typical superscalar processor on EE, EF and EC edges. This study also models a cache hierarchy, and therefore, instructions and loads that miss in the cache can suffer a large delay.
The study simulates each task as executing on its own core. Therefore, within a task, execution respects superscalar dependences such as fetching in-order. There can be instruction cache misses, which can lead to large latency on an FF edge, otherwise it depends upon the fetch width of the processor. Buffer stalls are also modeled within a task through CF edges.

10.1.3 Idealizations in the Study

Some of the constraints in a real SP system are not modeled in this study. The study idealizes commit bandwidth available to the system, therefore there is zero latency on in-order CC edges. This is done to allow for increases in commit bandwidth, or for other ways in which instructions might be committed such as bulk commit, etc.
Data-dependences that cross tasks don't suffer any penalty other than the execution unit latency. Similarly, the task spawn edge doesn't incur any delay. This is done to allow for advances in architectural mechanism that reduce cost to task spawn actions and inter-task dependence handling. In addition, if there is a cost associated with task-related actions, this creates a trade-off in task spawning, and therefore an optimal task selection policy is needed to accurately estimate the upside potential of speculative parallelization. Designing an optimal task selection policy is an NP-hard problem, because tasks interact with each other, and therefore an exponential number of task selections have to be tried out to find the optimal one. Therefore, this study removes the costs associated with spawning a task, and all possible task options are spawned as separate "tasks". The resulting potential therefore is quite optimistic, but it is an upper bound for the stated assumptions about the architecture.
In addition, the study allows for infinite number of tasks. A real system will have only a finite number of task resources, such as cores. However, modeling that constraint will again require optimal task selection to make the best use of limited resources, which is again NP-hard. Therefore this constraint is removed. Finally, this study allows tasks to be spawned in a nested manner, whereas in most SP research prototypes, tasks are spawned in-order and nesting is not allowed.

10.1.4 Results

Table 10.2 lists the parameters for an individual core used for this study. As identified above, the objective of this study is to impose the architectural constraints that are likely to exist in a SP architecture, without overly restricting the system. Therefore, in-order fetch with limited peak fetch bandwidth is enforced within a core. Cache and branch predictor sizes, as well as execution latencies are chosen to be representative of contemporary microprocessors. However, a large buffer space is modeled to allow for advancements such as early reclamation of resources which effectively increase the available buffer space for speculative instructions.

Parameter	Value
Fetch Width	4 instrs/cycle (per task)
Branch Predictor	8K-entry Combined, 8K entry gshare, 8K entry bimodal, 8K entry selector, 13 bits of history
Misprediction Penalty	10 cycles
Functional Units	4 identical general purpose units per task
L1 I-Cache	32Kbytes, 4-way set assoc., 128 byte lines, 10 cycle miss
L1 D-Cache	32Kbytes, 4-way set assoc., 64 byte lines, 10 cycle miss
L2 Cache	512Kbytes, 8-way set assoc., 128 byte lines, 200 cycle miss penalty
Reorder Buffer	64K entries
Scheduler	64K entries

Table 10.2: Parameters Used for the Study

Benchmark	Superscalar	Speculative Par. (SP)	SP without single BB loop
bzip2	1.45	15.86	13.89
crafty	2.39	24.39	24.39
gap	1.17	4.20	4.00
gcc	1.80	112.01	112.01
gzip	2.25	77.78	9.15
mcf	1.85	160.44	60.34
parser	1.17	6.32	6.29
perlbmk	1.66	2.04	2.04
twolf	1.47	44.26	44.26
vortex	2.78	477.55	477.55
vpr.place	1.90	16.05	16.05
vpr.route	1.78	103.60	103.60
ammp	1.91	24.15	23.54
applu	3.67	2405.17	2140.04
apsi	3.63	1671.59	275.30
art	1.64	584.48	9.88
equake	3.81	566.13	566.13
mgrid	3.77	3755.58	3755.30
swim	3.99	11572.73	4851.54

Table 10.3: IPC numbers for Upside Potential Study. Integer and Floating Point benchmarks are shown separately.

Figure 10.3 shows the performance potential of speculative parallelization, as predicted by this study. The results are depicted for a set of SPEC benchmarks, and these are classified into two categories: low parallelism benchmarks (figure 10.1) and high-parallelism benchmarks (figure 10.2). The actual IPC numbers for an individual superscalar core (with the same parameters as table 10.2) and the speculative parallelization limit are presented in table 10.3. Note that most of the floating point benchmarks have large amounts of parallelism, while most integer benchmarks

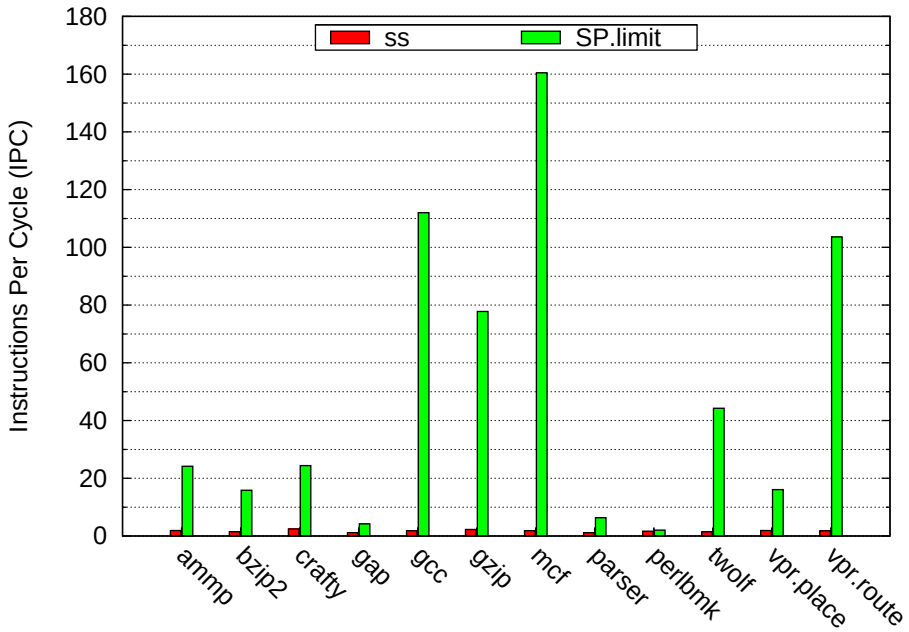

Figure 10.1: Benchmarks with low amounts of parallelism.

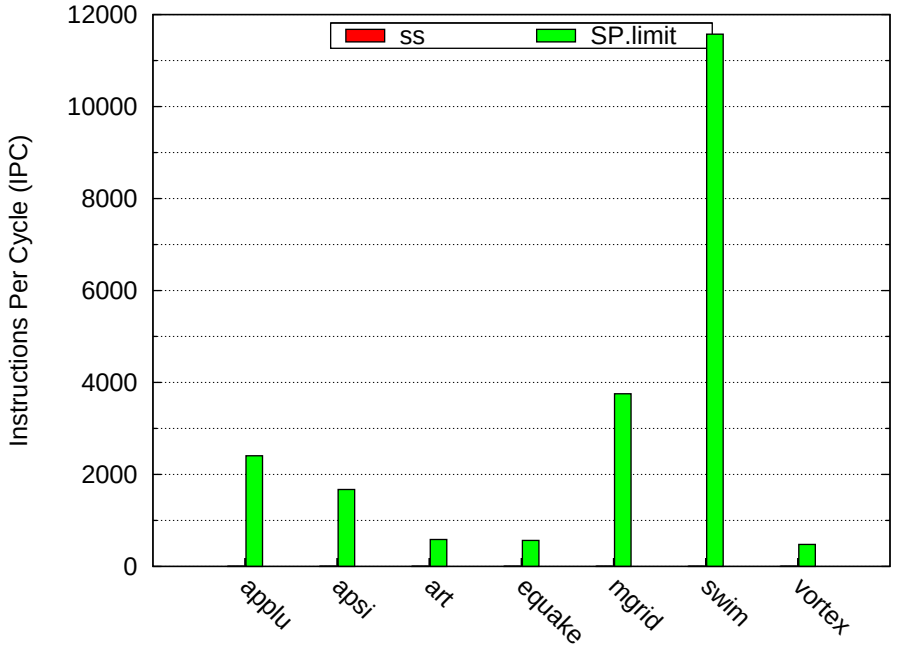

Figure 10.2: Benchmarks with low amounts of parallelism.

Figure 10.3: Upside potential of speculative parallelization.

offer low upside potential.

Also note that spawn points corresponding to loops that consist of just a single basic block are important. These are not captured by the basic-block level postdominator analysis. For single

basic-block loops the spawn points obtained from that analysis all jump out of the loop, and no postdominator corresponds to loop iteration spawn, so separate spawn points are added (note that these are also postdominators, just not postdominators of control transfer instructions). Table 10.3 shows that these spawn points are important for several benchmarks, especially for floating point benchmarks.

One could take this process to the limit, and do a postdominator analysis at the instruction level and spawn all tasks thus obtained. Each task in this case would comprise a single instruction. This is what Lam et. al did in their study. However, this strategy is unlikely to succeed in a real SP system because there are overheads associated with spawning a task, and so tasks comprising single instructions will likely degrade performance. Further, real systems have limited cores, so reasonably sized tasks that deliver large improvements in performance will be preferred over tiny tasks. Therefore this study restricts itself to basic block sized tasks at minimum.

The exception is made for loop cases, because in several cases (especially in Floating Point case), these loops can run for a long time, and not parallelizing the loop can create an Almdahl's bottleneck. Further, for several Floating Point benchmarks, the single basic block that makes up the loop body can be quite large (sometimes thousands of instructions) due to limited control flow. This could be generalized by breaking very large basic blocks into tasks, but at least for these benchmarks, no cases (outside loops) were found for that scenario.

10.2 Task Granularity and Parallelism

The study described in section 10.1 allowed a task to spawn a postdominator as a new task, irrespective of how far (in terms of dynamic instructions) the spawned point was from the spawner point. The SP system modeled allowed for a very large buffer so a large number of speculative instructions could be buffered until they were ready to be committed. Thus very large speculative tasks could be supported by individual cores in that study.

However, real SP systems have limited speculative buffers which also constrains the maximum allowed size of a task. Therefore such systems can't afford to spawn tasks into arbitrarily distant regions of the program. Another challenge to supporting large tasks is that the current techniques to detect data misspeculations and for inter-task data communication don't scale to large task sizes and large number of tasks.

Therefore, most SP systems are quite restrictive in the size of tasks allowed. This restriction also impacts the upside potential of speculative parallelization. This study investigates the impact of allowed task size on the upside potential of speculative parallelization. The setup is the same as before, with extremely large buffers (64K scheduler and reorder buffer), no inter-task data synchronization (true dataflow limits), and nested OOO spawning. The maximum allowed task size is varied and performance is tracked as a function of the task size.

Task size is determined through offline profiling. Since the size of individual instances of a task might vary due to effects like control flow, the offline profiling step measures the average size for a

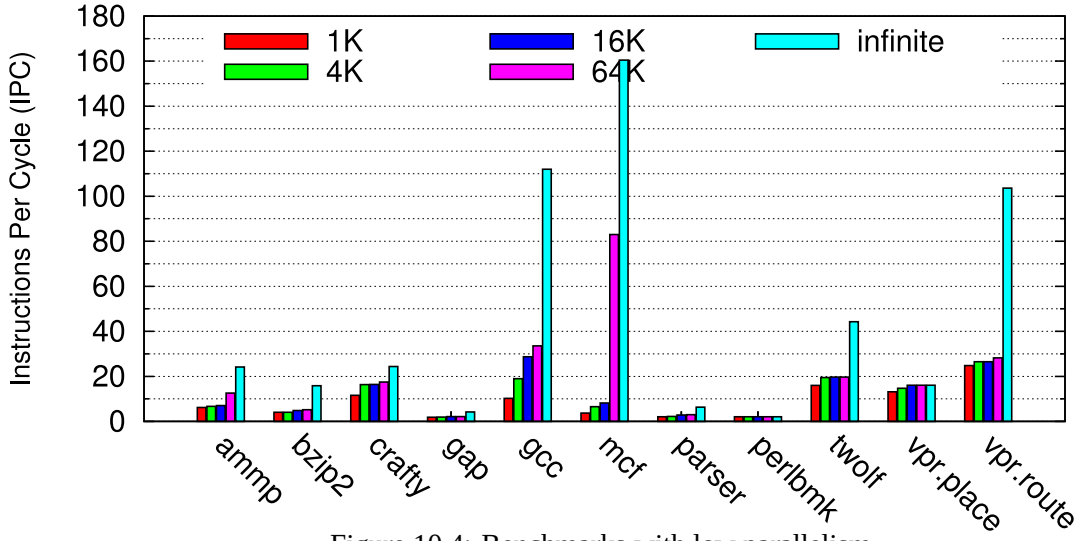

Figure 10.4: Benchmarks with low parallelism

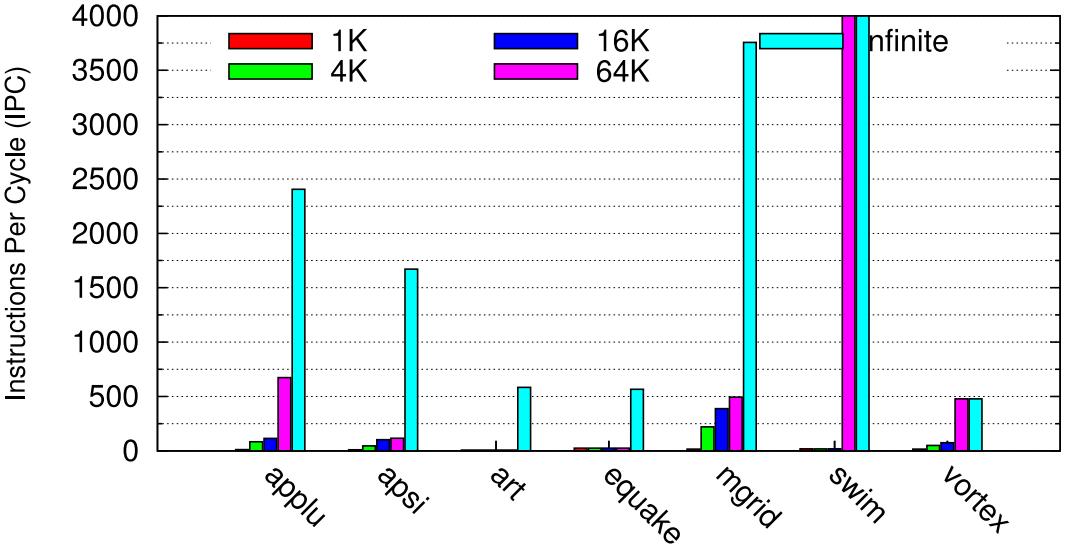

Figure 10.5: Benchmarks with large parallelism. Note that swim has a potential of 6492 at task size of 64K.

Figure 10.6: Impact of allowed task size on performance potential of speculative parallelization.

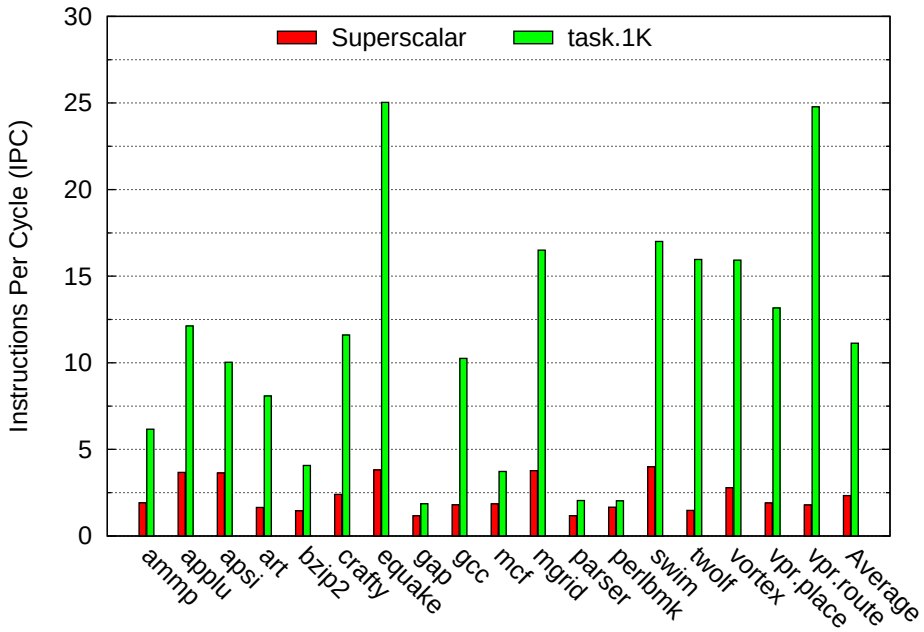

Figure 10.7: Potential of speculative parallelization at task size of 1K instructions, which represents a potential improvement of about 4x over a single superscalar core.

task as well as the maximum size. The size is defined as the distance (in terms of dynamic committed instructions) between the spawner and the spawnee instructions.

10.2.1 Results and Analysis

Figure 10.6 shows the performance potential as task size is varied, all the way from 1K instructions to 64K instructions. Note that these numbers represent the maximum allowed average task size allowed. In addition, a limit is placed on the maximum size of any dynamic instance of that task, which is 4 times the maximum average. Thus, if any dynamic instance of a task was profiled to be more than 4 times the allowed size limit, the task would be pruned out even though its average length was within the allowed size.

There are several interesting points to note. First, while the potential of speculative parallelization is huge, it is a function of the allowed task size. For most benchmarks, constraining tasks to be small severely limits the performance potential. Several benchmarks have a great amount of parallelism at large granularities, and limiting the task size to be small prevents that parallelism from being exploited for performance. For most SP research prototypes, the maximum allowed task size is around 1K instructions. Figure 10.7 shows the potential of SP at that task granularity, which is orders of magnitudes lower than the potential without that constraint. Therefore, SP systems should revisit this constraint to increase the achievable performance. Complexity-efficient mechanisms to support large speculative tasks and to perform inter-task data-dependence

management at larger granularity should be researched.
For some of the other applications (crafty, gap, parser, vpr.place, etc), primarily integer applications, on the other hand, most of the potential is at a small granularity. Current SP systems, therefore, are well-suited for these kind of applications. However, these applications have meager amounts of parallelism and cannot scale to large number of cores. To be truly effective, therefore, SP systems must find a way of parallelizing applications with large amounts of parallelism.
Another interesting aspect of these results is that parallelism doesn't increase uniformly with granularity. For several applications, there is a sudden large increase in parallelism once a certain granularity is allowed. For example, applu, swim, mcf, and vortex all see a large increase in the performance potential at task sizes of 64K. For swim, this increase is in several orders of magnitudes. For mgrid, there is a jump at granularity of 4K, after which there is a gradual increase in parallelism. This burstiness of parallelism has to do with the application structure. The next section gives an example for this effect.

10.2.2 Task Granularity in Swim

As figure 10.5 shows, the impact of allowed task granularity is pretty dramatic on *swim*. At sizes lower than 64K instructions, the benchmark has a very low upside potential (around IPC 17). However, at a size of 64K, there is a sudden jump in the IPC potential to around 6500, which represents an increase of around 400X. This section delves deeper into swim to understand the reason for this behavior.
Swim spends most of its time in doubly nested loops in very similarly structured (and behaving) functions *calc1*, *calc2* and *calc3*. Each of these functions contains a doubly nested loop where the inner loop goes around for 512 iterations for the input selected. The size of each inner loop iteration is around 120 instructions. Therefore, each outer loop iteration runs for around 62K instructions. The outer loop also goes around for 512 iterations.
In order to extract parallelism at the outer-loop granularity, therefore, the system needs to be able to spawn the next outer loop iteration as a separate task, which is around 62K instructions away. Therefore at task sizes below around 64K, only the inner loop is parallelized and leads to an IPC potential of around 17.
However, as soon as a task size of 64K instructions is allowed, it enables parallelization at multiple loop nests (512 outer loop iterations, each with 512 inner loop iterations). This leads to an explosion in the performance potential, which could keep a much larger number of cores busy. Note that the IPC potential is not as high as one would expect, because there are other serial regions which pose an Almdahl's bottleneck to parallel performance. In addition, inter-task loop index variable dependences also limit parallel performance. Transformations like strip-mining and unrolling could help reduce their overhead.
These results illustrate that applications can have parallelism at different granularities depending upon their structure such as size of loop nests, and depending upon which loop nest can be parallelized. Based upon the results of figure 10.5, it seems that benchmarks frequently have much

larger amounts of parallelism at outer loop nests compared to the innermost loop. It is important to be able to extract that parallelism for success of speculative parallelization.

10.3 Cost of Enforcing Inter-Task Data Dependences and Task Penalties

Parameter	Value
Inter-task data dependence	Synchronization policy 5 cycle communication penalty
Spawn Penalty	5 cycles
Reconnection Penalty	5 cycles

Table 10.4: Parameters Used for Task Spawning and Inter-task data dependences.

The studies of sections 10.1 and 10.2 optimistically assume that there will be no cost to spawning a task on a separate core. However, real speculative parallelization systems incur a cost whenever a task is spawned. Large costs come from enforcing data-dependences across tasks and from the extra penalty associated with spawning and merging tasks. This section tries to estimate the impact of these costs on the upside potential of speculative parallelization.

The setup used for this study is similar to the one in section 10.2. This study limits itself to tasks of average sizes of 1K dynamic instructions or less. This is constraint is enforced because that is the domain in which most current speculative parallelization systems operate. Individual cores are similar to the ones described in section 10.1, but the scheduler and reorder buffer modeled is of size 512 entries, which is an aggressive but reasonable point for current and near-future systems. Figure 10.8 illustrates the impact of these costs on performance compared to a limit study that idealizes on these costs. The “ooo.orcl” bar shows the performance when there are no costs to spawning a task. The “ooo.thresh_20.penalty” configuration synchronizes data-dependences as described in section 5.3.3. Thus, inter-task data-dependences with producers after the spawner point have to wait for the spawner task’s arrival at reconnection point before their produced value can become available to the spawned task. In addition, there is a penalty for task spawning and reconnection. Table 10.4 summarizes these costs.

Note that a task selection is needed when there are penalties involved for task spawning and inter-task data-dependences. This is because these costs can cause some tasks to actually degrade performance, and such tasks need to be pruned out. For this study, a task selection was made by placing a minimum threshold of 20 cycles gain per instance of a task as described in section 8.2. Therefore, the performance difference between the two bars in figure 10.8 can be attributed to two factors: 1) the cost from synchronization and task-related actions, and 2) untapped performance due to suboptimal task selection. Even though it cannot be proved, it is quite likely that the former factor is a major contributor to the difference observed above, because the above selection was found to be the best among a variety of thresholds on task behavior.

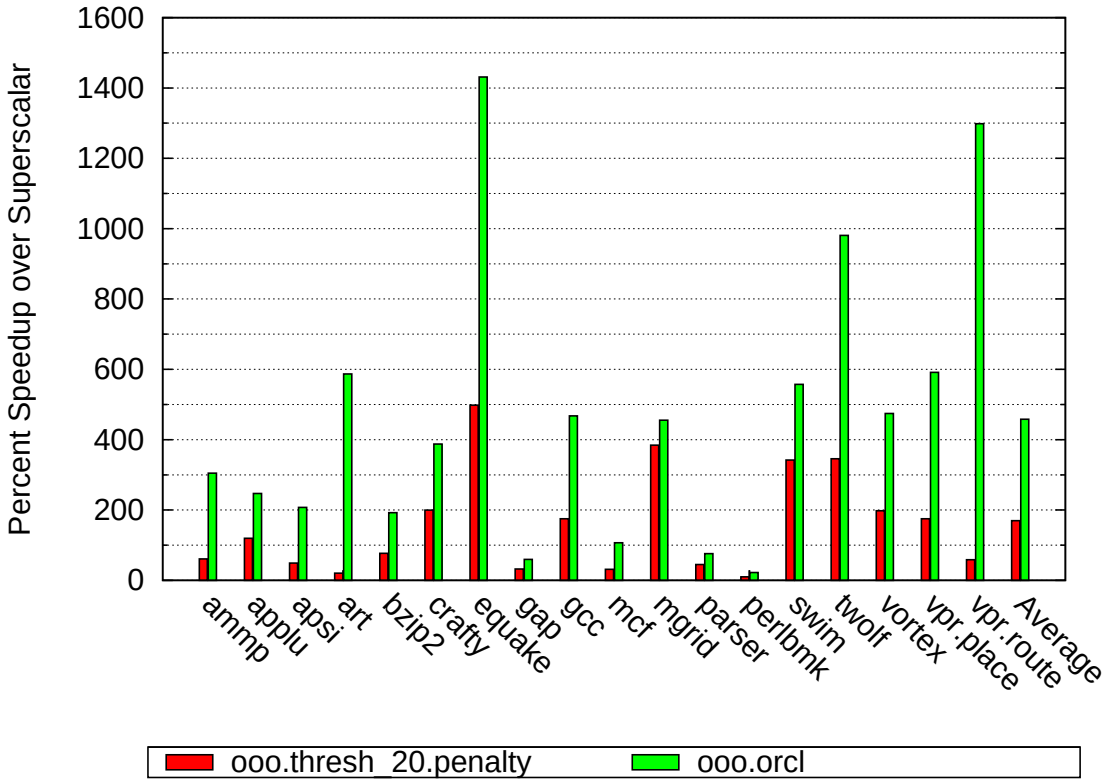

Figure 10.8: Impact of synchronizing data-dependences and imposing penalty for task spawn and reconnection.

There are several noteworthy points in these results. For benchmarks “vpr.place” and “vpr.route”, there is a large performance difference between the cases where inter-task data-dependences are not synchronized and where they are. This arises because the limit study optimistically sends over values as soon as they are produced in the earlier task. However, a real system might not know when the last write has occurred down the correct path until it is quite late. These benchmarks have a large number of low-confidence branches, so managing inter-task data-dependences will introduce large costs, irrespective of the policy used.

Another major effect is the addition of penalty for task spawning and reconnection and for synchronized inter-task data-dependences. This has the effect of making several tiny tasks unprofitable because the overhead of spawning and reconnecting tasks swamps out any benefit that might have been delivered by the task. In addition, tasks that have lot of incoming data-dependences also suffer costs when these dependences are synchronized. Sometimes, the producer might be several dynamic tasks away and multiple synchronization and communication delays might be incurred. This can cut the IPC performance quite dramatically. On the whole, data-dependences and task penalties bring down the potential of speculative parallelization by more than a half.

10.4 Nested Parallelism and Out-of-Order Task Spawning

The configuration modeled in section 10.1 allows tasks to be spawned in a nested manner, that is, out-of-order task spawning. Most speculative parallelization systems, on the other hand, model an in-order spawning system. This means that a task can only spawn one other task (that completes and retires) over its lifetime. Once a task has spawned off a later task that has not been squashed due to misspeculations, it can no longer spawn another task. This restriction greatly simplifies the system. The spawned task can be run on the adjacent core, and spawn and reconnect signals can be sent over a ring-type network. Determining program order is trivial.

However, in-order spawning prevents the exploitation of nested parallelism. The most straightforward example is that of a nested loop where parallelism exists both in the inner as well as outer loop. In-order spawning forces the system to choose between spawning either solely within the outer loop or only in the inner loop. Out-of-order spawning on the other hand can allow a task can do multiple spawns, as long as the task spawned later is nested within the earlier task. This ability frees the system from choosing which nesting level to spawn in, and potentially allow the ability to exploit parallelism at multiple nests. However, supporting it incurs large amounts of hardware complexity. Attempts have been made to propose simple solutions [46] but the problem is inherently harder.

It is easy to see that out-of-order spawning is required to be able to exploit all the available parallelism. For example, in the swim example of section 10.2.2, an in-order spawning system could spawn either in the inner loop (leading to an IPC potential of 17), or at the outer loop level (leading to a potential of 560). However, to exploit all the parallelism in that loop, tasks need to be spawned at both levels (leading to a multiplicative impact on the potential, which goes up to 6500). However, as identified by sections 10.2 and 10.3, the restrictions on task granularity and cost of data-dependences place significant constraints on the exploitable parallelism. In particular, the task granularity constraint in most speculative parallelization systems limits the maximum size of any individual task. This also makes it unlikely that there could be a large number of nested tasks possible. For example, most applications won't have more than two or three loop nests fit within 1K instructions. Further, even if there are multiple nested task spawn opportunities within a range of 1K instructions, an important question is whether there is exploitable parallelism in multiple nests given the cost from data-dependences and task penalties. In addition, most speculative parallelization systems have limited cores, so even if there is parallelism at multiple nesting levels, there may not be enough resources to exploit all of that parallelism. The objective of this section is to explore the potential for out-of-order spawning under these constraints.

This study explored (from a performance perspective) the potential benefits of OOO spawning against an optimized in-order selection. The in-order task selection policy used was described in section 8.2. A threshold of 20 was used for minimum per-instance benefit predicted by the model. Nesting analysis was incorporated for in-order selection. Out-of-order spawning, on the other hand, doesn't require a nesting analysis. Therefore, all tasks that exceeded a threshold of 20 (cycles gain per instance) were selected. The evaluation system synchronized inter-task data-dependences,

and imposed a 5-cycle penalty for task spawning, reconnection, and inter-task data communication.

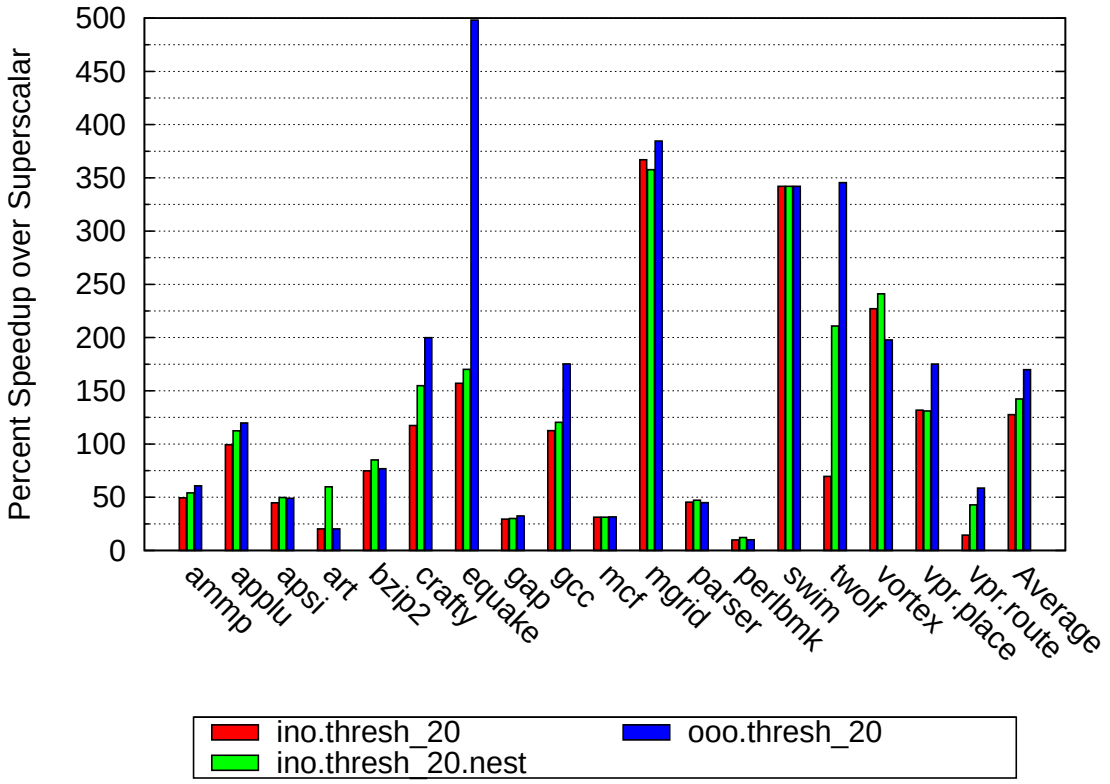

Figure 10.9: In-order spawning (without and with nesting analysis), compared to out-of-order spawning, on a system that has infinite cores, has latency for inter-thread data communication and synchronization.

Figure 10.9 shows the performance of OOO spawning on a system with infinite task units, when compared to in-order spawning (with and without nesting analysis). Three application behaviors are observed. The first set (e.g. ammp, apsi, bzip2, etc.) of applications don't benefit much from the ability to spawn out-of-order, indicating that there is not much nested parallelism in these benchmarks. By placing a threshold on task performance, a good selection can be isolated.
A second set of applications (applu, crafty, twolf and vpr route, etc.) has nested parallelism and therefore an upside from OOO spawning. An in-order task selection that just places a threshold performs significantly worse than its out-of-order spawning counterpart. However, incorporating nesting analysis in the in-order task selection helps bridge this gap quite successfully, taking the performance quite close to that of OOO spawning.
A final set of applications (equake, gcc, vpr.place, etc) have large amounts of nested parallelism all of which cannot be exploited by in-order spawning. For example, equake has a nested loop with parallelism in both nesting levels. The inner loop takes several data-cache misses causing buffer stalls. Further, the loop branch is hard to predict, which costs a large branch misprediction penalty

each time the loop is exited. To exploit parallelism, both inner loop iterations as well as the loop fall-through need to be spawned. Out-of-order spawning systems can do that. But in-order spawning systems have to choose between the two nesting levels.
Another point to note is that while there is a large performance potential gained from out-of-order spawning at infinite cores, there isn't much gain when the cores are limited. If smartly chosen in-order tasks can keep the resources busy, then there is no need to bring in the additional complexity of out-of-order spawning. This is observed even for benchmarks like equake, where there is enough parallelism in the outer nesting level to keep 4 cores busy.

10.5 Impact of Constraining Available Cores

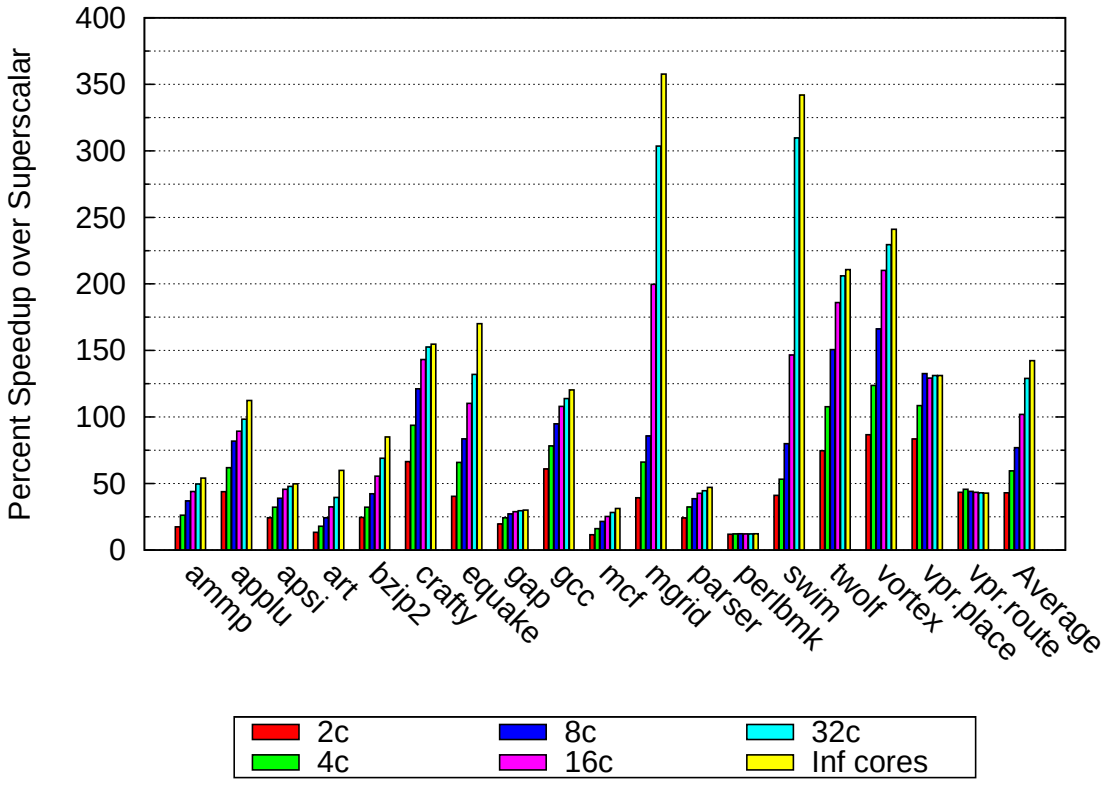

Figure 10.10: Impact of scaling cores on performance on a system that has latency for inter-task communication and performs inter-task data synchronization.

Evaluations in section 8.2 showed encouraging performance for speculative parallelization on a 4-core system. This section investigates the potential for more performance given a larger number of cores for spawning tasks, and how this performance scales. Figure 10.10 shows the speedup achieved as the number of cores are scaled from 2 to 32. The limit when infinite cores are available is also shown.

Several application behaviors are observed when scaling the number of cores. The largest class is applications that show limited additional returns when the number of resources are scaled, such as ammp, apsi, gap, parser, perlbmk, and vpr. For these applications, two or four cores is probably the right operating point. This is because these applications seem to have inherently parallelism at this granularity. Even with 4 cores the performance on these applications gets quite close to their upside performance potential identified in section 10.2. There are other applications which scale quite well even to 32 cores or beyond. This includes swim, mgrid and vortex, which display large amounts of loop-level or procedure-level parallelism. Finally there are applications that scale moderately but with largely diminishing returns, and it might be worthwhile to scale up to 8 cores if performance is the prime objective.

CHAPTER 11

CONCLUSIONS

The move towards multicore architectures has created a gap between the desire of programmers to continue programming productively in sequential programming models and the requirement of parallelizing applications to extract performance from the available cores. Technologies are needed to help bridge this gap without placing all the responsibility upon programmers.
Implicit Parallelization is one potential solution that builds upon the previously successful tradition of under-the-covers parallelization followed by superscalar architectures over the last decade or so. If successful, Implicit Parallelization allows programmers to think and write in sequential models, but still reap the benefits of additional cores through increased parallel performance.
However, Implicit Parallelization faces several roadblocks to its success, both at the level of architecture as well as from applications. A major challenge is identifying where profitable parallelism exists in applications, and partitioning it into tasks that can execute concurrently. Since sequential execution semantics have to be maintained, application control and data-dependences must be enforced. In addition, data-dependences can be hard to identify statically especially the dependences that occur because different memory accesses alias to the same location. Enforcing these dependences can add substantial cost to implicitly parallelized execution. These costs can have a large impact on the profitability of tasks, and make performance very sensitive to the quality of tasks selected for parallelization.

11.1 Thesis Summary

This thesis developed a novel approach of thinking about the challenges that arise in Implicit Parallelization. The insight is to approach the problem of finding and exploiting parallelism in terms of instruction criticality. In that framework, the potential for extracting parallelism in a region of program can be identified by the prevalence of fetch-criticality of instructions in that region. This thesis explored the reasons for fetch-criticality and formalized the notion of "Fetch-Criticality Generating Events" or FCGEs, that are responsible for causing fetch-criticality in superscalar execution, thereby creating the potential for exploiting parallelism.
The next step was to develop a formal model to represent program execution on an Implicit Parallelization architecture using a "dependence graph model". The model is built upon previous work for modeling superscalar execution. It is able to capture application as well as architectural dependences. It is also able to represent the costs and delays associated with task-related actions,

such as cost for managing data-dependences that cross task boundaries. The model can be used to construct a dependence graph for a particular execution of a program, and find the “critical path” of execution as well as the slack on various dependence edges. It is also a useful tool to help understand and quantify the performance trade-offs that arise in Implicit Parallelization, such as deciding whether or not to spawn a potential task.

Next, this thesis developed a quantitative model to predict the performance benefit of spawning a potential task on an Implicit Parallelization system. The model is based on an analysis of the application trace. It was validated to be accurate for several benchmarks. This model was then used to drive the task selection policy for the Polyflow Implicit Parallelization architecture. Polyflow belongs to a class of Implicit Parallelization architectures known as “Speculative Parallelization” architectures because it can speculate upon ambiguous data-dependences. The task selection policy based on the above model was found to significantly outperform other policies used in previous work. An important insight in making the task selection was that for in-order spawning systems (such as Polyflow and most other previous architectures in this domain), it is important to account for nesting relationships between tasks to select tasks at the most profitable nesting level.

Next, this thesis looked at the broader picture by exploring how the potential for performance can be enhanced by relaxing some of the previous constraints imposed: both at application level and at the architectural level. At the application level, criticality analysis is once again useful in finding top bottleneck data-dependences that limit parallelism. This thesis finds, encouragingly, that several of these dependences are not “essential” to the computation. Rather they were “accidental” due to unfortunate implementation choices made by the programmer. These dependences can be removed easily without changing application semantics significantly. In other cases, essential dependences sometimes limit parallelism.

This thesis developed a tool called SPARTAN that can point programmers to important bottlenecks to parallelism so that they can be refactored on a priority basis. The tool was validated by finding top bottleneck data-dependences in several applications. This thesis went one step further and actually removed an accidental dependence in one benchmark. The impact of removing the dependence was quite close to what was predicted by the model. This tool combines nicely with the task selection tool to form an iterative flow for Implicit Parallelization.

Finally, the thesis revisited architectural design decisions made by most Implicit Parallelization systems to understand the disparity between upside potential and achieved performance. There were some interesting results. The thesis found that decisions like limited task sizes severely limit the scope of Implicit Parallelization systems. On the other hand, other choices like not allowing for nested task spawns doesn’t impact performance much once other constraints are in place while saving on architectural complexity. Overall, the thesis suggests encouraging directions for moving forward towards making Implicit Parallelization a more successful approach.

REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.

[2] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[3] B. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," *Int'l Symp Computer Architecture*, vol. (ISCA-28), pp. 74–85, 2001.

[4] A. Glew, "Mlp yes! ilp no!," in *Wild and Crazy Ideas Session, 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[5] K. Malik, M. Agarwal, S. S. Stone, K. Woley, and M. I. Frank, "Branch-mispredict level (blp) parallelism for control independence architectures," *Int'l Symp. High Performance Comp. Arch.*, vol. (HPCA-14), 2008.

[6] D. W. Wall, "Limits of instruction-level parallelism," DEC Western Research Laboratory, Research Report 93/6, Nov. 1993.

[7] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," *Int'l. Symp. Comp. Arch.*, vol. (ISCA-19), pp. 46–57, 1992.

[8] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the Pentium^{TM}4 processor," in *Intel Technology Journal*, 2001, no. 1.

[9] E. S. Tune, D. M. Tullsen, and B. Calder, "Quantifying instruction criticality," in *In 11 th International Conference on Parallel Architectures and Compilation Techniques*, 2002, pp. 104–113.

[10] B. Fields, R. Bodík, and M. Hill, "Slack: Maximizing performance under technological constraints," *Int'l Symp Comp Arch*, vol. (ISCA-29), 2002.

[11] B. Fields, "Using criticality to attack performance bottlenecks," Ph.D. dissertation, University of California at Berkeley EECS Department, 2006.

[12] J. D. Wiest and F. K. Levy, *A Management Guide to PERT/CPM*. Prentice-Hall, 1974.

[13] S. T. Srinivasan and A. R. Lebeck, "Load latency tolerance in dynamically scheduled processors," in *Journal of Instruction Level Parallelism*, 1998, pp. 148–159.

[14] S. T. Srinivasan, A. R. Lebeck, R. D. ching Ju, and C. Wilkerson, "Locality vs. criticality," *Computer Architecture, International Symposium on*, p. 0132, 2001.

[15] B. R. Fisk and R. I. Bahar, "The non-critical buffer: Using load latency tolerance to improve data cache efficiency," in *In IEEE International Conference on Computer Design*, 1999, pp. 538–545.

[16] B. Calder, G. Reinman, and D. M. Tullsen, "Selective value prediction," in *In 26th Annual International Symposium on Computer Architecture*, 1999, pp. 64–74.

[17] E. Tune, D. Liang, D. M. Tullsen, and B. Calder, "Dynamic prediction of critical path instructions," in *In Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 185–196.

[18] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn, "Interaction cost and shotgun profiling," *ACM Trans. Archit. Code Optim.*, pp. 272–304, 2004.

[19] R. Pyreddy and G. Tyson, "Evaluating design tradeoffs in dual speed pipelines," in *Workshop on Complexity Effective Design*, 2001.

[20] J. Seng, E. Tune, and D. Tullsen, "Reducing power with dynamic critical path information," *Intl Symp Microarchitecture*, vol. (MICRO-34), pp. 114–123, 2001.

[21] P. Salverda and C. Zilles, "A criticality analysis of clustering in superscalar processors," *Intl Symp Microarchitecture*, vol. (MICRO-38), pp. 55–66, 2005.

[22] C.-Q. Yang and B. Miller, "Critical path analysis for the execution of parallel and distributed programs," *Distributed Computing Systems, 1988., 8th International Conference on*, pp. 366–373, Jun 1988.

[23] J. K. Hollingsworth, "Critical path profiling of message passing and shared-memory programs," *IEEE Trans. Parallel Distrib. Syst.*, pp. 1029–1040, 1998.

[24] T. Li, A. R. Lebeck, and D. J. Sorin, "Quantifying instruction criticality for shared memory multiprocessors," in *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, 2003, pp. 128–137.

[25] R. Nagpal and A. Bhowmik, "Criticality driven energy aware speculation for speculatively multithreaded processors," in *International Conference of High-Performance Computing*, vol. 12, Dec 2005, pp. 19–28.

[26] J. Tuck, W. Liu, and J. Torrellas, "CAP: Criticality analysis for power-efficient speculative multithreading," *Intl Conf Computer Design*, vol. (ICCD), 2007.

[27] C.-Y. Cher and T. N. Vijaykumar, "Skipper: A microarchitecture for exploiting control-flow independence," *Int'l. Symp. Microarchitecture*, vol. (MICRO-34), pp. 4–15, 2001.

[28] A. S. Al-Zawawi, V. K. Reddy, E. Rotenberg, and H. H. Akkary, "Transparent control independence (TCI)," *Int'l Symp Comp Arch*, vol. (ISCA-34), 2007.

[29] A. Hilton and A. Roth, "Ginger: Control independence using tag rewriting," *Int'l Symp Comp Arch*, vol. (ISCA-34), 2007.

[30] E. Rotenberg and J. E. Smith, "Control independence in trace processors," *Int'l. Symp. Microarchitecture*, vol. (MICRO-32), pp. 4–15, 1999.

[31] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," *Int'l Symp Computer Architecture*, vol. (ISCA-22), pp. 414–425, 1995.

[32] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," *Int'l Symp Computer Architecture*, vol. (ISCA-27), pp. 1–24, 2000.

[33] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun, "The Stanford Hydra CMP," *IEEE Micro*, vol. 20, no. 2, 2000.

[34] V. Krishnan and J. Torrellas, "A chip multiprocessor architecture with speculative multithreading," *IEEE Transactions on Computers*, vol. 47, September 1999.

[35] H. Akkary and M. A. Driscoll, "A dynamic multithreading processor," *Int'l Symp. Microarchitecture*, vol. (MICRO-31), pp. 226–236, 1998.

[36] M. Agarwal, K. Malik, K. M. Woley, S. S. Stone, and M. I. Frank, "Exploiting postdominance for speculative parallelization," *Int'l Symp. High Performance Comp. Arch.*, vol. (HPCA-13), pp. 295–305, 2007.

[37] P. Marcuello, A. González, and J. Tubella, "Speculative multithreaded processors," *Int'l. Conf. Supercomputing*, vol. (ICS-12), pp. 77–84, 1998.

[38] I. Park, B. Falsafi, and T. N. Vijaykumar, "Implicitly-multithreaded processors," *Int'l. Symp. Comp. Arch.*, vol. (ISCA-30), pp. 39–51, 2003.

[39] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," *High Perf. Computer Arch.*, vol. (HPCA-7), pp. 37–48, 2001.

[40] J.-Y. Tsai, Z. Jiang, and P.-C. Yew, "Compiler techniques for the superthreaded architectures," *Int. J. Parallel Program.*, vol. 27, no. 1, pp. 1–19, 1999.

[41] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, July 1987.

[42] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[43] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "POSH: A TLS compiler that exploits program structure," *Principles and Practice of Parallel Programming*, vol. (PPoPP-11), pp. 158–167, 2006.

[44] A. I. Moshovos, "Memory dependence prediction," Ph.D. dissertation, University of Wisconsin-Madison Computer Sciences Department, 1998.

[45] A. KleinOsowski and D. Lilja, "Minnespec: A new spec benchmark workload for simulation-based computer architecture research," 2002.

[46] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas, "Tasking with out-of-order spawn in TLS chip multiprocessors: microarchitecture and compilation," in *19th Int'l Conf. Supercomputing (ICS)*, 2005, pp. 179–188.

[47] K. Malik, M. Agarwal, and M. I. Frank, "Adaptive memory synchronization (ams): Balancing the risks and benefits of inter-thread load speculation," *Second Annual Reconfigurable and Adaptive Architecture Workshop*, vol. (RAAW-2), 2008.

[48] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *25th International Symposium on Computer Architecture (ISCA-25)*, June 1998, pp. 142–153.

[49] K. Malik, “Critical branches and lucky loads in control independence architectures,” Ph.D. dissertation, University of Illinois at Urbana Champaign Electrical and Computer Engineering Department, May 2009.

[50] A. Roth, “Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization,” in *ISCA 32*, 2005, pp. 458–468.

[51] A. Garg, M. W. Rashid, and M. Huang, “Slackened memory dependence enforcement: Combining opportunistic forwarding with decoupled verification,” in *ISCA ’06: Proceedings of the 33rd annual international symposium on Computer Architecture*, (Washington, DC, USA), IEEE Computer Society, 2006, pp. 142–154.

[52] K. Malik, “Confidence based out-of-order register renaming for dynamically multithreaded processors,” M.S. thesis, University of Illinois Department of Electrical and Computer Engineering, Dec. 2006.

[53] M. W. Hall, T. J. Harvey, K. Kennedy, N. McIntosh, K. S. McKinley, J. D. Oldham, M. H. Paleczny, and G. Roth, “Experiences using the parascope editor: an interactive parallel programming tool,” *SIGPLAN Not.*, pp. 33–43, 1993.

[54] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[55] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam, “Suif explorer: an interactive and interprocedural parallelizer,” in *PPoPP ’99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1999, pp. 37–48.

[56] C. Zilles and G. Sohi, “Understanding the backward slices of performance degrading instructions,” 2000, pp. 172–181.

[57] C. Zilles and G. Sohi, “Execution-based prediction using speculative slices,” 2001, pp. 2–13.

[58] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, “The stampede approach to thread-level speculation,” *ACM Trans. Comput. Syst.*, vol. 23, no. 3, pp. 253–300, 2005.

[59] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry, “Compiler optimization of scalar value communication between speculative threads,” *Arch. Support Prog. Lang. Operating Sys.*, vol. (ASPLOS-X), pp. 171–183, 2002.

[60] M. K. Chen, “Test: A tracer for extracting speculative threads,” in *In The 2003 International Symposium on Code Generation and Optimization*, IEEE Computer Society, 2003, pp. 301–312.

[61] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai, “A cost-driven compilation framework for speculative parallelization of sequential programs,” *Prog. Lang. Design and Implementation*, vol. (PLDI), pp. 71–81, 2004.

[62] S. Wang, X. Dai, K. S. Yellajyosula, A. Zhai, and P. chung Yew, “Loop selection for thread-level speculation,” in *In Proceedings of the 18 th International Workshop on Languages and Compilers for Parallel Computing*, 2005.

[63] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee, "Compiler support for speculative multithreading architecture with probabilistic points-to analysis," *Principles and Practice of Parallel Programming*, vol. (PPoPP-9), pp. 25–36, 2003.

[64] J. Tubella and A. González, "Control speculation in multithreaded processors through dynamic loop detection," *High Perf. Comp. Arch.*, vol. (HPCA-4), pp. 14–23, 1998.

[65] P. Marcuello and A. González, "A Quantitative Assessment of Thread-level Speculation Techniques," *Int'l. Parallel and Distributed Proc. Symp.*, vol. (IPDPS-14), pp. 595–604, 2000.

[66] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," *Arch. Support Prog. Lang. Operating Sys.*, vol. (ASPLOS-VIII), pp. 58–69, 1998.

[67] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," *Arch. Support Prog. Lang. Operating Sys.*, vol. (ASPLOS-XI), pp. 107–119, 2004.

[68] T. N. Vijaykumar and G. S. Sohi, "Task selection for a multiscalar processor," in *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, 1998, pp. 81–92.

[69] T. N. Vijaykumar, "Compiling for the Multiscalar architecture," Ph.D. dissertation, University of Wisconsin-Madison Computer Sciences Department, Jan. 1998.

[70] P. K. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton, "Single-program speculative multithreading (SPSM) architecture: compiler-assisted fine-grained multithreading," *Conf on Parallel Arch and Compilation Techniques*, vol. (PACT-1), pp. 109–121, 1995.

[71] P. Marcuello and A. González, "Thread-spawning schemes for speculative multithreading," *High Perf. Comp. Arch.*, vol. (HPCA-8), pp. 55–64, 2002.

[72] A. Bhowmik and M. Franklin, "A general compiler framework for speculative multithreaded processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 8, pp. 713–724, 2004.

[73] C. G. Quinones, C. Madriles, J. Sanchez, P. Marcuello, A. Gonzalez, and D. M. Tullsen, "Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices," *Prog. Lang. Design and Implementation*, pp. 269–279, 2005.

[74] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, "Min-cut program decomposition for thread-level speculation," *Prog. Lang. Design and Implementation*, vol. (PLDI), pp. 59–70, 2004.

[75] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, "Speculative decoupled software pipelining," in *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, IEEE Computer Society, 2007, pp. 49–59.

[76] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke, "Uncovering hidden loop level parallelism in sequential applications," in *In Proc. of the 14th International Symposium on High-Performance Computer Architecture*, 2008.

[77] M. K. Prabhu and K. Olukotun, "Exposing speculative thread parallelism in spec2000," in *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM Press, 2005, pp. 142–152.

[78] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August, “Revisiting the sequential programming model for multi-core,” in *MICRO 40*, 2007.

[79] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” *Prog Lang Design and Impl*, vol. (PLDI), June 2005.

AUTHOR’S BIOGRAPHY

Mayank Agarwal was born in New Delhi, India, on July 14, 1982. He graduated from Indian Institute of Technology Delhi in 2004 with a Bachelor of Technology degree in Computer Science and Engineering. He completed the Master of Science degree in Computer Science at the University of Illinois in 2006. After completing his PhD, he will be joining Microsoft as a Software Development Engineer.