COLLECTING AND REPRESENTING PARALLEL PROGRAMS WITH HIGH PERFORMANCE INSTRUMENTATION

A Thesis
Presented to
The Academic Faculty

by

Brian Paul Railing

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

Georgia Institute of Technology
December 2015

COLLECTING AND REPRESENTING PARALLEL PROGRAMS WITH HIGH PERFORMANCE INSTRUMENTATION

Approved by:

Professor Thomas M. Conte, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

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

Dr Bruce Worthington
Windows Platform Health
Microsoft Corporation

Professor Santosh Pande
School of Computer Science
Georgia Institute of Technology

Date Approved: 30 October 2015

Man's chief end is to glorify God.

ACKNOWLEDGEMENTS

A dissertation is not a sprint, but a marathon. I have trained for years with many people, practiced my work, and then find so many there at the race cheering me on until I reach the finish. Many have contributed to this research, from the anonymous reviewers and readers who have given feedback and suggestions, to the family, friends, and church members encouraging and praying for me. I am thankful for their efforts and contributions that have helped carry me to the finish.

The research group is vital to completing the dissertation work, with their encouragement and collaboration, and the serious discussions over the core of so many topics. I would like to thank Eric Hein, whose efforts were vital to get Contech off of the ground and whose patience for my speculation, questions, and venting lasted throughout this work. Sriseshan Srikanth, thank you for providing the late push to complete this research. And thank you to Dr. Jesse Beu with whom I discussed much in those early years, for whom we started preparing Contech to help us understand parallel programs, and to quote your dissertation, "we will publish together!"

To my advisor, Dr. Tom Conte, who took me in when I was an advisor-less graduate student, thank you. Also thank you for telling us over and over again that we needed to think bigger when first planning Contech; we just wanted to solve a problem, but you pushed the topic from a paper to a dissertation. And a further thank you for supporting my passion and desire to teach, letting me take those opportunities to develop and grow in my chosen profession.

To Dr. Bruce Worthington, who helped form my knowledge of computing in the real world as my manager for four years and now served on my dissertation committee,

thank you. Thank you to the rest of my committee for their time and effort in serving the future generation of scientists, guiding and reviewing us.

A deep thank you to my parents, the teacher and the computer scientist, who inspired me in my chosen career, read so many drafts over the years, and helped support this work in so many ways.

I would be remiss if I did not thank my son, Titus, who would tell me, "Daddy, type!" For being only two years old, you understand so much of what is going on around you and what I needed to be doing.

Finally, Vickie Railing, who agreed to marry me and travel across the country to enter graduate school. You are my helper each and every day. For how much you have read of Computer Science, listened to me discuss my work, my reviews, and everything else, this is your degree too.

TABLE OF CONTENTS

DEDICATION . . . . . **iii**

ACKNOWLEDGEMENTS . . . . . **iv**

LIST OF TABLES . . . . . **x**

LIST OF FIGURES . . . . . **xi**

SUMMARY . . . . . **xiii**

1 INTRODUCTION **1**

1.1 Definitions 2

1.2 Future Challenges 3

1.2.1 Architectural Challenges 3

1.2.2 Programmer Challenges 4

1.3 Designing a New Tool 6

1.4 Contributions 7

1.5 Organization 8

2 TASK GRAPH REPRESENTATION **9**

2.1 Extending from Prior Task Graphs 9

2.2 Parallel Program Representation 10

2.3 Contech Task Graph 12

2.4 Visualizing a Contech Task Graph 16

2.5 Incorporating a Parallel Programming Model into the Task Graph Representation 17

2.6 Example Task Graphs 18

2.6.1 Why are syncs (i.e. non-work) a unique type of task 20

2.7 Other Parallel Program Representations 23

2.7.1 Communicating Sequential Processes 23

2.7.2 Concurrent Collections 24

2.7.3 LogP 25

2.7.4 Legion . . . 25
2.7.5 Galois . . . 25
2.7.6 X10 . . . 26
3 PROGRAM INSTRUMENTATION . . . 27
3.1 Reducing Instrumentation Overhead . . . 28
3.1.1 Inlined Instrumentation . . . 29
3.1.2 Sampling . . . 29
3.1.3 Mirroring Program . . . 30
3.2 Source Instrumentation . . . 31
3.2.1 GProf . . . 31
3.2.2 ICECAP . . . 32
3.2.3 LLVM-Based Approaches . . . 33
3.3 Instrumentation Frameworks . . . 34
3.3.1 Pin . . . 35
3.3.2 Valgrind . . . 37
3.3.3 DynamoRIO and PiPA . . . 38
3.3.4 PEBIL . . . 39
3.3.5 QEMU . . . 39
4 CONTECH INSTRUMENTATION . . . 41
4.1 Overall Design Goals . . . 41
4.2 Front end . . . 43
4.2.1 Front End Instrumentation . . . 44
4.2.2 Event Ordering . . . 49
4.2.3 Runtime Buffer Management . . . 52
4.2.4 Trace Formats . . . 58
4.3 Middle Layer . . . 58
4.4 Instrumentation Performance . . . 61
4.4.1 Data Write Performance . . . 63

4.4.2 Background Writing Performance 66
4.5 Running on ARM ISA 70
4.6 Comparison with Pin 71
4.6.1 Instruction Execution Rate 73
4.6.2 Data Cache Miss Rates 75
4.7 Variation on Workload Parameters 76
4.7.1 Increasing Input Size 77
4.7.2 Thread Counts 78
4.8 OpenMP Support 79
4.8.1 Rodinia Benchmarks 81
4.8.2 LULESH 82
4.9 MPI Support 82
4.10 Cilk Support 83
4.11 Other Implementation Issues 84
4.12 Contech Compiler Wrapper 85
4.12.1 Undefined Behavior 86
4.13 Predictor of Contech Performance 87
5 PRACTICAL TASK GRAPH ANALYSIS 90
5.1 Back end tools 90
5.1.1 Types 92
5.1.2 Determinism in Analysis 94
5.1.3 Basic Blocks 95
5.2 Parallel Programming Paradigms 96
5.3 Sample Analyses 97
5.3.1 Control Flow Graphs 98
5.3.2 Cache Simulation 98
5.3.3 Memory Usage 99
5.4 Summary 100

6 PARALLEL PROGRAM MODELING 101
6.1 Related Work 102
6.1.1 Modeling Parallel Programs 102
6.1.2 Simulating Parallel Programs 102
6.1.3 Software-based Reconfiguration 103
6.1.4 Hardware-based Reconfiguration 105
6.2 Reconfigurable Parallel Architecture 105
6.2.1 Reconfiguration Time 107
6.3 Architectural Model 109
6.3.1 Integration with Program Model 110
6.3.2 Modeling Communication and Coherence 110
6.4 Parallel Program Modeling 114
6.5 Experimental Results 116
6.5.1 Performance Improvement 119
6.5.2 Context Phase Variation 121
6.5.3 Comm Factor Granularity 122
6.5.4 Staticness of Configurations 123
6.5.5 Overlapping Reconfiguration and Wait Times 125
6.6 Summary 126
7 CONCLUSION 128
REFERENCES 130

LIST OF TABLES

1 Instrumentation Tools with Similar Functionality to Contech (x - Support Present, b - Basic Block count without Full Path Information) . 28

2 System Configuration for Contech Measurements 43

3 Static Global Accesses per Benchmark 65

4 System Configuration for Contech Measurements on ARM 70

5 Runtime Overhead for PARSEC (simsmall) with Clang and Optimized Contech (without lto) on ARM 70

6 Decision Forest Regression Model Parameters 87

7 Baseline Function Unit Configuration 106

8 Baseline Resource Configuration 106

9 Data Race Analysis Results (in Bytes) 112

10 Benchmark Characteristics 117

11 Number of Operations Concurrently Issued in Blackscholes Critical Task (1:0) 119

12 Staticness of Ideal Per-Task Configurations 124

LIST OF FIGURES

1 Different Simulation Types in Computer Architecture, (Figure 1 from Rico, et al. [102]) . . . 7

2 Aggregating a Computation Graph into Tasks (Work: 1, 3, and 4; Create: 2) . . . 11

3 Classic Task Graph, circa 1993 [22] . . . 12

4 Parallel Program Graph [107] . . . 13

5 Legend for Visualized Contech Task Graphs . . . 15

6 Contech Task Graph Features by Type . . . 16

7 Simple OpenMP Program as a Contech Task Graph . . . 19

8 Simple OpenMP Program Using OMP Tasks . . . 20

9 Code Coverage by Contech Instrumentation . . . 45

10 Runtime Instrumentation Design (grey instructions are original code) 48

11 Instrumentation Slowdown due to Varying the Threshold for Using Small Buffer Copy Path . . . 53

12 Impact of Changing Buffer Size on Contech Runtime Slowdown . . . 54

13 Cost to Allocate Buffers based on the Size of Buffer . . . 55

14 Varying Buffer Allocation and Reuse . . . 57

15 Comparison of Contech and Pin Runtime Slowdown for PARSEC and SPLASH (simmedium, 16 threads) and NAS (A, 16 threads) . . . 61

16 Overhead Increase from Runtime Instrumentation over Uninstrumented Programs . . . 62

17 Frequency of Byte Values in Contech Trace . . . 68

18 Simple Compression Algorithm . . . 69

19 Average Increase of Architectural Events over Uninstrumented Execution 73

20 Correlation between Runtime Slowdown and the Change in the L1I Cache Miss Rate . . . 74

21 Relationship between Native and Instrumented L1 D Cache Store Miss Rates . . . 76

22 Size of Event List Generated from PARSEC and SPLASH with Different Inputs . . . 77

23 Runtime Slowdown of Contech Instrumented PARSEC and Splash 2 workloads by Varying the Number of Threads 78

24 Contech Instrumentation Performance on the Rodinia Benchmark Suite 81

25 Task Graph Recursively Computing the 6th Fibonacci Number using Cilk 83

26 Approximate Fitting of Runtime Slowdown based on Other Metrics . 88

27 Average and Median Basic Block Cycle Times 95

28 Average Static versus Dynamic Memory Operations per Basic Block . 96

29 Subset of Dynamic CFG from Fluidanimate (BBID and Execution Count) 97

30 Miss Rate for Selected PARSEC, SPLASH-2, and NAS benchmarks with 16 threads, across shared caches sized 1KB to 256MB (2^{10} to 2^{28}) 98

31 Estimated Working Set versus Actual Resident Set Sizes for Uninstrumented Benchmarks in MB 99

32 Proposed Reconfigurable System Workflow 101

33 Overlapping Reconfiguration and Synchronization 108

34 Simulating Communication with Randomness 113

35 Using task graphs to drive parallel program modeling and analysis . . 115

36 Speedups of ROI in PARSEC Benchmarks 118

37 Speedups of ROI in PARSEC Benchmarks Accounting for Buffer Increases 120

38 Configuration Type Used in Accelerating the ROI in PARSEC Benchmarks (Green - Integer, Red - Floating Point, Blue - Memory, Black - Minimal Speedup) 121

39 Speedups from Varying Comm Factor Granularity in PARSEC Benchmarks (2X All Configuration) Using Earlier Architectural Model . . . 122

40 Impact of Reconfiguration Latency (in cycles) on Speedup 125

SUMMARY

Computer architecture has looming challenges with finding program parallelism, process technology limits, and limited power budget. To navigate these challenges, a deeper understanding of parallel programs is required. I will discuss the task graph representation and how it enables programmers and compiler optimizations to understand and exploit dynamic aspects of the program.

I will present Contech, which is a high performance framework for generating dynamic task graphs from arbitrary parallel programs. The Contech framework supports a variety of languages and parallelization libraries, and has been tested on both x86 and ARM. I will demonstrate how this framework encompasses a diversity of program analyses, particularly by modeling a dynamically reconfigurable, heterogeneous multi-core processor.

CHAPTER 1

INTRODUCTION

Modern hardware is going through a paradigm shift. The solutions and techniques of the past decades are reaching physical limits. Already architectures are embracing parallelism to continue to provide improved performance. Many opportunities are looming ahead, such as 3D stacking, heterogeneous components, and dark silicon. Designing for these opportunities requires greater understanding of the software.

Software is ever increasing in size and complexity. To continue its development and improvement, programmers must understand their software. How does the software work with the compiler, hardware, operating system, and other programming libraries. To understand the software and how it is executing, tools and analyses are required. Without quality tools and analyses, software will not be able to fully utilize the resources provided, especially as the computing ecosystem continues to evolve and change.

Therefore, there is a continuing need of programmers, compilers and computer architects to understand modern parallel programs, and each in different ways [7]. The common approach has been to develop a targeted instrumentation and corresponding analysis for each particular need. In many circumstances, existing instrumentation frameworks have been available to assist in collecting the specific data about the program, mitigating some of the work to develop the instrumentation. While some

> This work will demonstrate that a common representation, in the form of a task graph, can provide a unified platform for a diversity of parallel program analysis, while matching the performance overheads of more targeted approaches.

instrumentation is designed to have negligible overhead (e.g., using sampling), allowing these tools to be used with production systems, there is still a need for more detailed insight into parallel programs.

Some problems of analysis can be solved with simple tools and techniques, yet to encompass the great diversity of common analyses, more comprehensive approaches are required. The task graph representation is one such approach. By gathering the threading interactions of the software along with the trace of execution, a task graph can serve as a common representation upon which analysis tools can be developed.

Task graphs of various kinds are well established in the study of efficient scheduling of parallel tasks [76] [109] [120] [132], as well as evaluating future architectures [4] [42] and parallelizing applications [52]. Yet task graphs have been limited by the nature of their generation. In software, research has relied on programs being appropriately annotated or written in a task-oriented language. Even hardware-based approaches have assumed the programs have been extended to expose the "tasks". Task graph representations have generally been designed for targeted problems of scheduling, which necessitates extending the representation if the task graph is to be used to solve other complex problems.

1.1 Definitions

Task graphs will be described in detail in Chapter 2. The concept is that parallel programs are composed of sequential execution regions, called tasks. Tasks are delimited by operations that may affect the concurrency, such as exposing additional concurrency or coordinating between tasks.

Concurrent operations are those that may potentially operate simultaneously with other operations in a program or system. The architecture of the system, scheduling decisions, and other constraints then determine whether any two operations actually

execute in parallel with each other. A task graph is expressing the actual concurrency exposed and thus potential parallelism of a program.

Parallel paradigms are different approaches to expressing the concurrency of a program. The approaches may be implemented through runtime or library APIs, compiler pragmas or extensions, or integral to the programming language.

Heterogeneous architectures are those whereby the processing elements of a single system differ in some aspect, such as the frequency or cache configuration. Heterogeneity can have both static and dynamic traits, for example, the ARM big.LITTLE [50] combines different processor cores (static) and dynamically enabling and frequency scaling the cores. Scheduling and modeling such a system will be analyzed in Chapter 6.

1.2 Future Challenges

Software development is being guided by the changes in computer architecture, whereby processor designs are constrained by their power and energy requirements. This constraint brings increasing focus on core counts and heterogeneity as ways for the hardware to continue to provide more performance. Software and programs are also burdened by their own success, whereby the designs and software architecture continue to grow, while layers of abstraction are introduced to aid the programmer. These burdens further distance the developer from the tools to leverage the changes in the hardware.

1.2.1 Architectural Challenges

What are the architectural challenges of the near future? Even now, the presence of multicore chip designs has created difficulties for compilers and programmers to achieve the performance and stability expected of modern software. In focusing on

increasing core counts, modern software does not yet have sufficient scaling to always use the cores available [41]. Simply increasing the core count will leave greater portions of the processor unused.

For the past several decades, the decreasing feature size of silicon has had similar decreases in power consumption; however, as the features increasing depart from classical mechanics and are subject to quantum effects, the savings in power are diminishing [41] [133]. Should the power consumed by the transistors exceed the thermal dissipation capability of the design, some of the transistors must be disabled at all times except in short bursts. This design would now have different on chip components that are enabled depending on the requirements of the executing code. Even switching to other cooling techniques cannot prevent the fundamental scaling problems of power dissipation.

Improvements in process technology have been shrinking the features in silicon; however, there is a looming limit in the size of features. Eventually, each feature would be the size of a single atom, at which point the transistor counts can only be increased by macro scaling, either larger dies or vertical stacking. There is always the possibility of changing materials, yet unless the fundamental architecture is changed the trends remain.

The above trends lead to increasing opportunity for die space to be devoted to different purposes, whether it be cores with a variety of computation power or specialized accelerators. In this heterogeneous space, how a program is designed to use the resources is of importance and requires more focused analyses.

1.2.2 Programmer Challenges

Programmers must ensure the correctness of their code, while meeting the architectural challenges that are proliferating the number of cores. One increasingly common approach is the use of alternative programming languages that better address

the costs and difficulties of modern programming. For example, Facebook recently adopted Haskell for better scalability in detecting spam messages [87]. Other groups have developed new languages, such as X10, Go, and Rust, or adopt different parallel paradigms, such as OpenMP or Cilk, in an effort to ease the burden on the programmer to meet these challenges. However, rewriting millions of lines of legacy code is rarely an option and therefore programmers also require improved support from tools.

Program correctness requires locks, barriers, and other forms of synchronization. Significant research has explored more efficient and scalable implementations both from software as well as hardware changes. Vallejo, et al. [118] provides a good summary of the various lock-based implementations available. It can be difficult for a programmer to make an informed decision as to the appropriate lock implementation based on the usage of the lock and the underlying architecture. Program scalability is further impacted due to coarse-grained locking, whereby a critical section protects more than is required for correctness. In many cases, it is possible to statically identify operations that do not require the critical section [11], and thereby reduce the size of the critical section. Furthermore, synchronization may also be split into finer-grained units to protect smaller sets of locations, thereby reducing contention when the different sets can be accessed independently.

By incorrectly reducing the critical sections and otherwise changing the synchronization, accesses to shared data may no longer be safely ordered and serialized, thereby introducing data races. Data races can be difficult to identify due to the diversity of synchronization in programs [116], as well as the impact of the memory model [73]. Changes to synchronization may also introduce deadlocks and other failing execution scenarios.

Programmers utilize the available abstractions in selecting algorithms and data structures during development to improve their productivity. These conventions may not be efficient from an architectural or compiler standpoint. One example is the use

of data structures in a program. *Brainy* [69] showed that not only do programmers often chose a suboptimal data structure, but the optimality of the choice can depend on the architecture itself. This work was extended to also consider the parallel usage of the data structures [90].

1.3 Designing a New Tool

The general approach by programmers to understand and analyze a program follows repeated iterations across increasingly complex tools toward the specific problem of interest. The complexity of the tools requires greater time to collect as well as analyze the information from the program. This has lead to a significant diversity of analysis tools.

The original challenge presented three years ago was identifying the thread and communication relations to best exploit Manager Client Pairing (MCP) [17] [18], which is a hierarchical coherence compositional design. Without any immediate tool available to solve the problem, a new instrumentation tool or plug-in would be required. The minimum data necessary for this problem would be the memory accessed by the parallel program, as well as the thread ID or context of each access. This diversity of data required to support MCP leads the design of what would become Contech to collect additional data in order that the tool would be able to support many of the existing tools and analyses. Given the desire to support this diversity, Contech's design generates traces of the program's execution, so that multiple tools can analyze precisely the same events from that execution.

Figure 1, from Rico, et al. [102], illustrates the decreasing use of trace-driven models for computer architecture. Program traces have difficulties modeling parallel programs, due to the need to capture timing related aspects of execution such as lock ordering. Fitting with this trend, only 16% of papers in ISCA 2015 used traces. The

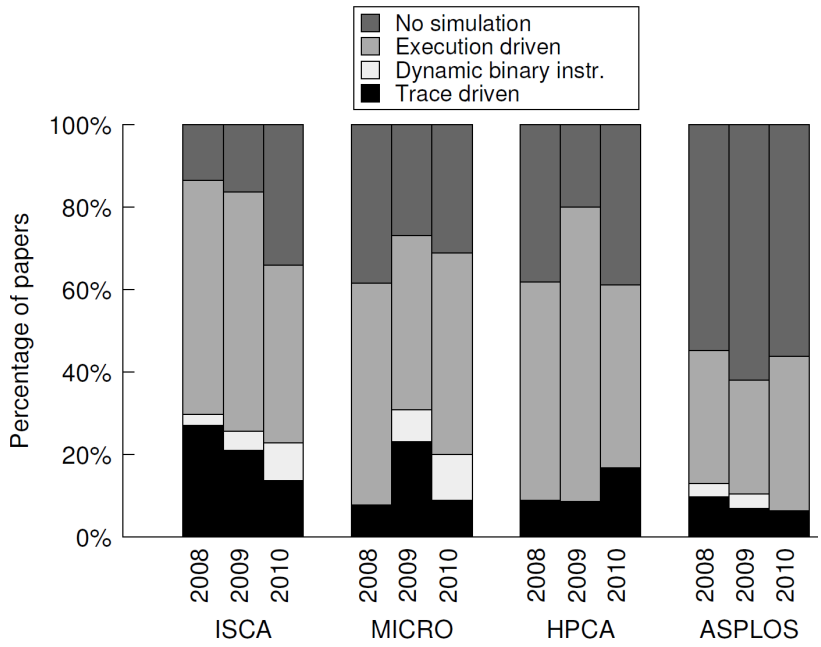


Figure 1: Different Simulation Types in Computer Architecture, (Figure 1 from Rico, et al. [102])

development of a new tracing tool would need to properly encapsulate the different timing aspects of parallel programs.

From interviews with parallel programmers [7], a regular concern with tools and debugging approaches is whether the techniques used to find the performance issue will introduce other phenomena (i.e., probe effects) that will mask the original issue. Therefore, any tool used to understand programs, especially parallel ones, must be designed with an awareness of the performance costs of its associated instrumentation.

1.4 Contributions

The contributions of this dissertation are three-fold:

- A novel, "Contech", task graph representation of parallel programs that supports independence of architecture, language, and threading paradigm. This representation splits synchronization tasks from work tasks, and includes the memory and execution trace.

- The design and implementation of a framework in the popular LLVM compiler infrastructure to efficiently generate the dynamic Contech task graph representation for arbitrary parallel programs.

- Demonstration of the comprehensive nature of the task graph representation by modeling the parallel programs on a parallel, reconfigurable architecture.

1.5 Organization

The remainder of the dissertation is organized as follows: Chapter 2 describes the task graph representation, Chapter 3 explores the different approaches taken to instrument and analyze programs, Chapter 4 demonstrates the Contech framework that collects and supports the analysis of task graphs. Chapter 5 will cover the analysis of task graphs, along with several sample tools. Chapter 6 presents a parallel reconfigurable architecture and shows how Contech task graphs can be used to model the architecture. Chapter 7 concludes the dissertation.

CHAPTER 2

TASK GRAPH REPRESENTATION

This chapter will present the theoretical underpinnings of the task graph representation that the Contech parallel program instrumentation and analysis framework uses. It will also discuss the basis and history of the task graph representation and how Contech's model differs from the prior work. These differences are what enables Contech task graph analysis to address a greater diversity of problems, as well as to encompass many parallel programs.

Parts of this chapter have been extended from the prior publication of Contech [100].

2.1 Extending from Prior Task Graphs

Contech is not the first work to propose task graphs for representing the parallel program and generate them from instrumented programs. However, prior approaches have been limited to specific languages and program structures, or required that programmers add annotations to their code so that a tool could construct a task graph. Nabbit [3] and Vandierendonck, et al. [120] both extend Cilk with annotations to identify tasks for scheduling, while Heumann, et al. [56] proposed similar annotations for Java to provide tasks with safety guarantees. Even hardware-based approaches to task scheduling have relied on programmer annotations made to C or C++ programs [42] [52] [76].

Adve, et al.[1] demonstrated a framework that can collect task graphs without any programmer annotations, when the program is written in High Performance Fortran (HPF). When they extended their work to more general programs, they relied on hand-generated task graphs, as they noted, "[we] manually wrote scripts to generate

each of the task graphs" [2]. Thus they decided to focus their work on an interesting analysis of parallel programs rather than extending their prior framework.

Tareador [8] used Valgrind-based instrumentation to generate a (artificial) task graph for the program. This task graph is primarily a recording of the sequential execution of the program, with additional user-supplied annotations to identify the potential parallelism of the program. It is primarily designed to explore parallelism strategies before implementation.

While not generating a task graph, Varuna [109] identifies virtual tasks out of parallel programs, using existing parallelism APIs to identify when parallel work is created, and dynamically finds efficient schedules of the running programs. ParaOps [102] [103] has been used to encapsulate parallel operations in execution traces, to better support parallel simulation. Varuna and ParaOps, along with Contech, represent parallel operations via abstraction to either better schedule, simulate, or analyze parallel programs.

The following section describes the task graph representation, as based out of prior publications. These papers developed the representation for theoretic analysis of scheduling algorithms, where the programs had been abstracted as task graphs.

2.2 Parallel Program Representation

A parallel program can be represented as a set of tasks that may execute in parallel and interact via both actions and data dependencies. A task graph models the behavior and execution of a parallel program as a directed acyclic graph, where nodes are tasks and edges are the explicit dependencies between tasks, which gives the following expressions for a task graph:

$G = (V, \overrightarrow{E}), \forall v \in V :< task >$

$\forall \overrightarrow{e} \in \overrightarrow{E} :< scheduling\ dependencies >$

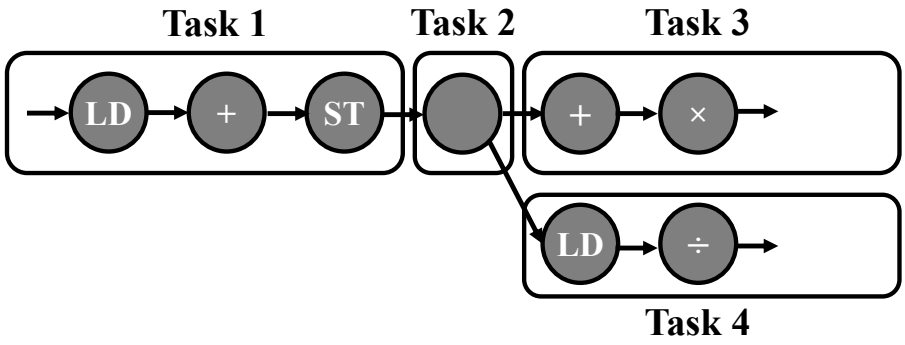


Figure 2: Aggregating a Computation Graph into Tasks (Work: 1, 3, and 4; Create: 2)

The focus of Contech has generally been on shared-memory programs; however, a distributed-memory program could be similarly represented where all memory accesses are mapped to disjoint sets of addresses and there exist send and receive actions that allow accesses between these sets [46], see also Section 4.9.

To derive a task graph, first begin with a program's execution, which is considered as a computation graph. In the computation graph, every node is an action, be it an instruction, basic block, function, or some other unit of execution. Some actions have additional explicit non-data dependencies on prior actions, and are termed *synchronization actions*. Programs may use these dependencies to enforce an order on its data dependencies. The actions are then aggregated together into tasks, thereby transforming a computation graph into a dynamic task graph. Figure 2 shows an aggregation of compute nodes into tasks.

The task graphs representation was first used to analyze program scheduling by decomposing the program into tasks [21] [22] [46]. Figure 3 shows an example task graph from that work. In it, tasks are circles and there are three types of edges: horizontal - continue, vertical - spawn, and curved - data-dependency. Additional research extended task graphs with their potential interactions [84] and found subsets of the graph with no interaction and representing the subset as a single node [15].

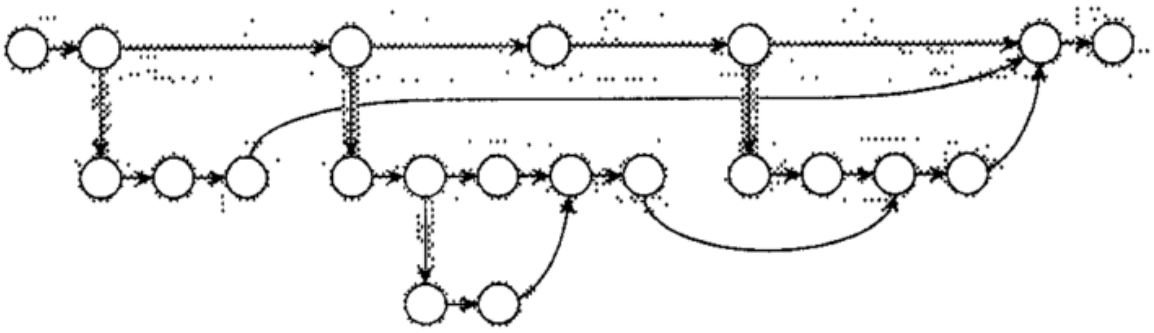


Figure 3: Classic Task Graph, circa 1993 [22]

Parallel Program Graphs [107] is an extension of control flow graphs (CFGs), which adds a parallelism node and synchronization edges; however, it does not cleanly partition out other edge types controlling parallelism. Figure 4 provides one example of a Parallel Program Graph. Note how the synchronization edges (S2 to S5, and S4 to S5) are only conveying the ordering information. This graph also demonstrates its CFG origin with the backward edge and representing each statement as a separate node. Finally, node S5 is a generic node and does not have a special type, even though it represents the end of the parallel section.

A task is defined as containing either a sequence of actions taken without an intervening synchronization action, or the synchronization action itself, which is similar to Cilk's definition of a strand [83]. Tasks consisting of the former are identified as work tasks (task partition W), which may contain individual instructions, basic blocks, or entire functions. In practice, most tasks are sequences of basic blocks; however, a standard system function such as `malloc` is stored intact in a task, although other work have used different sequences [21] [22]. Tasks may also contain a set of data locations accessed.

2.3 Contech Task Graph

One extension of Contech task graphs from prior work is that tasks containing synchronizing actions are assigned into one of four partitions on the task graph: creates

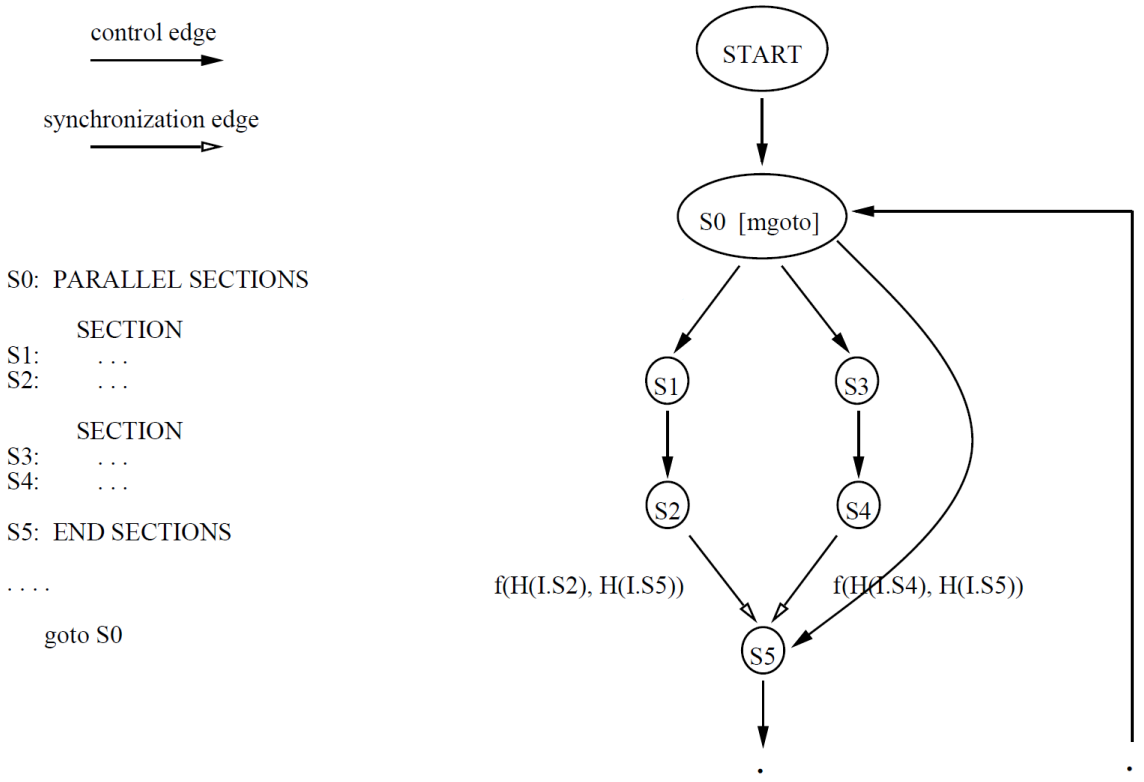


Figure 4: Parallel Program Graph [107]

(C), joins (J), syncs (S) and barriers (B), which along with the fifth partition, work (W), cover all tasks in the graph. Each *synchronization action* can only map to a single partition type in the task graph. Thus the task graph formalism is extended:

C, J, S, B, W are partitions on V:$< task\_type >$

These synchronization tasks define the topology of the task graph, and cleanly separate the synchronization from the useful work within the graph. Using these partitions, the task graph has one type of edge, and the cause of dependency edges is clear from the partitions of the tasks, whereas prior task graph representations either stored the cause as an action in the tasks or as different edge types. Thus the dependencies in the task graph are only those actions explicitly invoked by the program to order otherwise concurrently executing contexts. Unlike prior work, the dependencies

between tasks are not associated with data being produced or consumed by the tasks. In practice, the particular mapping of parallelism functions to the partitions is left to tools that generate task graphs. For example, the end of an OpenMP parallel region has an implicit barrier followed by a joining of the parallel execution, which could be mapped to both a barrier and join or just a join task.

The following properties are expected of each partition regardless of the architectural details of the generating tool: Create tasks increase the possible parallelism in the task graph, commonly having an *out* degree greater than their *in* degree. Join tasks are the complement of creates, in that they reduce the possible parallelism in the task graph. Sync tasks capture actions that impose an order between tasks, but without changing the parallelism in the task graph. The sync task may encapsulate a semaphore or condition variable, where one task is waiting on another task's notification. Tasks in the sync partition can also represent lock acquires and releases. The particular mapping is again left to tools implementing the task graph definition and could capture other functionality. Barrier tasks are similar to sync tasks, except that the ordering requirement is all to all rather than one to one (e.g., lock) or one to many (e.g., condition variable). Together, the dependencies in a task graph establish a partial order between the tasks. While it may be possible to create a total order using additional annotations such as the start and end times of tasks, such an order would be an artifact of a particular execution.

The set of work tasks can also be partitioned into disjoint subsets, where each subset represents a *context*. This partitioning of work tasks can be constructed using information known for each execution action. The practical definition of a context depends on threading models, but in general such a subset in the task graph represents the implicit data dependencies that many architectures carry between work tasks via registers, stack locations, return values, et cetera. As such, tasks in a context are

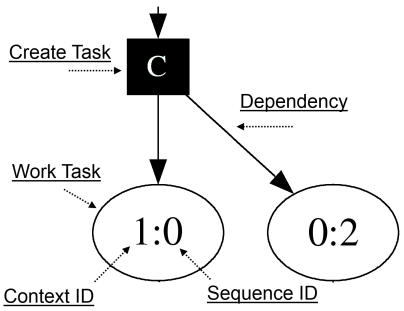
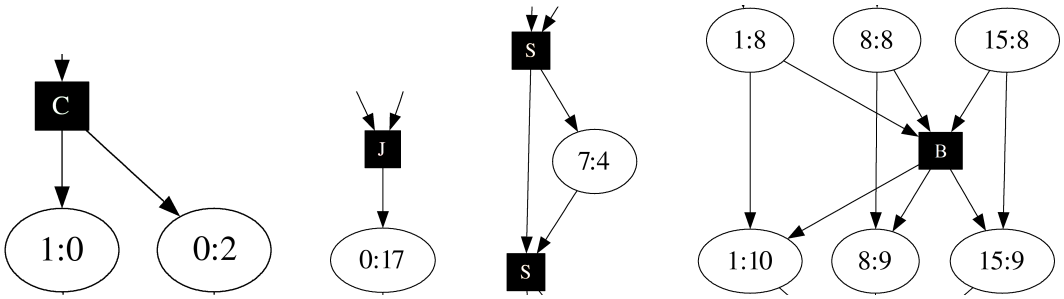


Figure 5: Legend for Visualized Contech Task Graphs

expected to depend on prior tasks in the context and may not be able to execute in parallel with each other, even if there are no explicitly represented data dependencies.

The task graph only contains the program order of the actions in each task, which may differ from the order dictated by the architecture executing the program. This property of the task graph also holds for any data locations accessed by the task. Dependencies between these actions may restrict the set of possible reorderings that the compiler or hardware can exploit. As the task graph is independent of the architecture, it does not preserve the actual execution order, only the program order. These orderings, along with the dependencies from synchronizing actions, define possible orderings of the actions in a parallel program. A correct parallel program uses synchronizing actions to restrict the set of possible orderings to those that preserve the intended dependencies between memory writes and reads in different threads. In other words, the synchronization actions prevent data races.

Contech work tasks contain sequences of execution and memory actions. The Contech task graph also contains properties and annotations such as, a set of predecessor and successor tasks for each node, as well as start and end times. Being annotations, the start and end times are not used in constructing the task graph, and are only provided for certain analyses.



(a) Create (b) Join (c) Sync (d) Barrier

Figure 6: Contech Task Graph Features by Type

2.4 Visualizing a Contech Task Graph

It is useful to visualize the elements of Contech task graphs. Figure 5 provides the legend for the subsequent figure. Tasks are uniquely labeled with a Task ID, which is the pair of <Context ID>:<Sequence ID>. The Context ID is a unique identifier for a context, akin to a thread ID. The Sequence ID is a monotonically increasing identifier in each context. Work tasks are represented by ovals, with their Task ID visible. Other task types have a Task ID, but are represented instead by their type. The edges in the graph are represented as arrows, with the direction of the arrow pointing towards the dependent task. Synchronization actions are black squares labeled with a single letter indicating the type. Note that while the implementation also assigns identifiers to these actions, they are not shown in the visualization. While these images are taken from actual task graphs, in our experience most parallel programs use thousands to millions of tasks during execution, which exceeds the capabilities of even a simple graphical visualization.

Figure 6 shows each of the task types from a task graph. The Create task (Figure 6.a) shows one task entering the create task and two tasks dependent on the create. The 1:0 shows that Context 1 has been created, and therefore Context 0 invoked the create. Join (Figure 6.b) has two tasks entering the join task, and one task (Context 0) continuing. Syncs (Figure 6.c) involve two tasks entering and leaving

the sync, which represent the two chains of dependencies. One chain of dependencies follows the sync tasks, as each sync must occur after its predecessor, for example, a lock is acquired after being released. The second chain of dependencies reaching a sync task follows the context that executes that sync, whether it be an atomic, lock, or other construct. The final task graph feature, the barrier (Figure 6.d), involves many tasks entering and leaving. Tasks after the barrier are dependent on both their context's last task as well as the barrier task itself, although the former is known transitively via the latter.

2.5 Incorporating a Parallel Programming Model into the Task Graph Representation

The process of integrating a new parallel programming model is straightforward. Each construct within a parallel programming model must be mapped to the appropriate representation in a task graph. The construct is considered for how it affects the (potential) parallelism of the program: is it increased (create), decreased (join), or has the parallelism remained the same (sync / barrier). The intent of the programmer also influences classification; for instance, atomic instructions can be used for fine-grained concurrent access to shared locations and should therefore be represented explicitly as sync tasks rather than be contained within work tasks.

When a representation for a construct is established, the construct must be appropriately instrumented to record the necessary information to represent the task. For create and join tasks, this often requires identifying the contexts involved in the operation. Sync and barrier tasks are identified by the address of the dynamic instance of the construct. This instrumentation is at the core of the Contech framework.

Condition variables, part of the POSIX threads standard, provide a way for threads to signal both 1:1 and 1:many. The core design is centered on representing condition wait, which performs three operations: it unlocks the mutex, after some time the condition variable is signaled, and the mutex is reacquired. Each of

the three operations is represented as a separate sync task performed on either the mutex or condition variable, with the mutex's or the condition variable's identifier (e.g., address) stored in their respective tasks.

OpenMP 3.0 introduced the *task* keyword, which was extended in 4.0 to add a *depend* keyword. As the OpenMP task increases the potential parallelism, it is first bracketed by Contech create and join tasks. Then each OpenMP task dependency provides an ordering constraint between separate tasks without changing the program's exposed parallelism, and each is therefore represented by a Contech sync task for the associated item. The constraint's identifier (e.g., address) are then used to link the dependencies together in the final task graph.

2.6 Example Task Graphs

In its simplest, a task graph is a single task. This program can be as short as "Hello World" or a multi-day serial architectural simulation. Across this range of serial programs, the programs are without any parallelism actions, and so the aggregation from the computation graph will never encounter an action that should be in a separate task, which contrasts with the example in Figure 2.

The program's computation graph can be initially viewed as being aggregated as a single task. Parallel actions are then selected that are part of larger tasks. Each parallel action (P), in general, will split the task (T) containing it into three pieces: computational actions in T prior to P, P itself, and the computational actions in T following P. Specific parallel actions may introduce additional tasks. A create action is part of the three split tasks plus one additional task per created context. And a barrier action splits each waiting context and introduces the single barrier task as a common dependency.

In practice, parallel programs will either be dominated by syncs and barriers or creates and joins, based on the underlying programming paradigm. In pthreads and

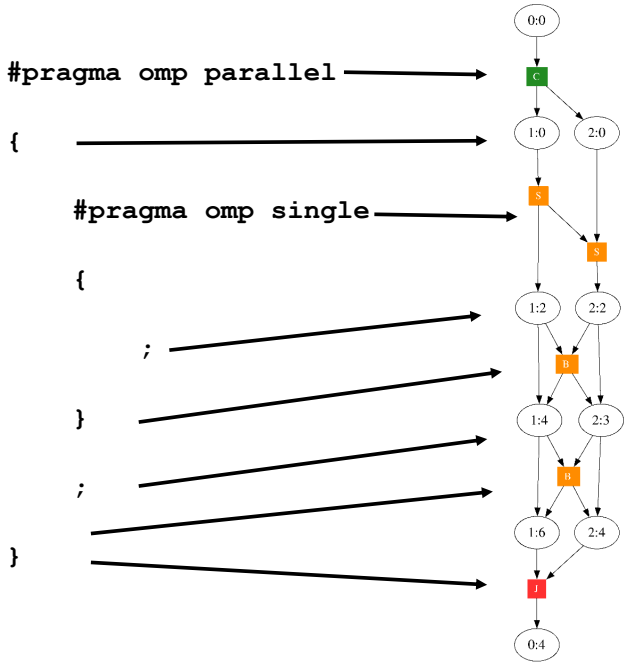


Figure 7: Simple OpenMP Program as a Contech Task Graph

MPI, threads are explicitly managed, therefore the high cost of thread creation results in few creates. Concurrent access by the parallel program to its data is protected through syncs (mutexes, condition variables, et cetera) and barriers. With OpenMP and Cilk, the programs will instead decompose the problem into parallel components that are created through `#pragma omp parallel` or `cilk_spawn`, which the parallelism runtime may map to different threads. Decomposing into finer granularity can then mitigate the need to use fine-grained synchronization. This analysis will be revisited in Section 5.2.

Figure 7 shows how a simple OpenMP program maps to a Contech Task Graph. From top to bottom, the parallel region creates two parallel contexts, which then execute up to the *single* directive. The contexts synchronize and one enters the region, while the other skips to the end of the region. The end of the *single* construct implies a barrier within this parallel region. The contexts continue executing, until they reach the end of the parallel region, where they wait at the implicit barrier and then join back into the serial execution. Note again that the implementation of the

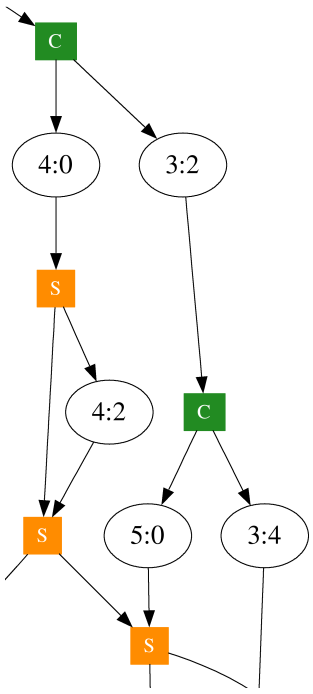


Figure 8: Simple OpenMP Program Using OMP Tasks

instrumentation used to generate this Contech task graph represented the implicit barrier and join separately.

Figure 8 demonstrates how the OpenMP task construct is represented in Contech, along with conveying the dependency information between tasks. Context 3 creates two tasks (context 4 and 5) each with an input and an output dependency on the same variable. The dependencies are each represented with a sync task, as each (potentially) orders the execution of different contexts. In the example figure, the syncs surrounding work task 4:2 are the input and output dependencies to that task. The OpenMP task in context 5 syncs on context 4's output after work task 5:0.

2.6.1 Why are syncs (i.e. non-work) a unique type of task

Prior work has generally treated non-sequential dependencies as separate edge types. For example, Blumofe, et al. [22] had three edge types: continuation, spawn, and data dependency. Parallel Program Graphs [107] used two edge types: control and

sync. Yet the Contech task graph only uses one edge type. More importantly, Contech promotes syncs and barriers to be distinct types of nodes, rather than using types of edges. In principle, this decision begins with the underlying computation graph representation whereby a synchronization would be one or more actions in the graph. Beyond this underlying representation, this section will work through several cases where prior designs would have difficulties adequately supporting desired functionality.

In general, synchronizations form a continuous chain throughout the program, as each synchronization is ordered with respect to the one before and the one after it, such as atomic operations on a single memory location. From this primitive synchronization, programmers (and architects, parallelism libraries, et cetera) can build more complex forms of synchronization, such as locks, condition variables, or semaphores. Some complex types may have implicit orderings that overlap with the underlying ordering. For instance, consider a lock that alternates between *acquire* and *release*. *Release* will follow *acquire* in the same context, and therefore will have the context's ordering constraint. In contrast, the *acquire* after a *release* may cross between contexts and require the constraint explicitly. If only the latter were required, then an edge from the releasing task to the acquiring task would convey this same ordering. In such a case, the synchronization action itself could be placed before (for *acquire*) or after (for *release*) the work of the task itself. Initially, Contech's design attempted to do that and place the synchronization with the work.

Three problems limit the suitability of including synchronization with work: identifying syncs in the graph, representing non-lock types, and measuring the costs of syncs. Tasks with syncs could be identified by extra flags, special edge types or even the number of incoming / outgoing edges. Differentiating the diversity of syncs can be (and is) accomplished by including an enum with the task. However, non-lock types do not have the clear *acquire* and *release* that enables inclusion with work.

Until now, the discussion of syncs has focused on locks; however, as initially noted, there are a variety of types built on atomic primitives. If the sync is not a lock, then there may be no implicit ordering and therefore the complete order should exist in the task graph. Thus, the synchronization event (*current*) has two edges: (*previous* $\rightarrow$ *current*) and then (*current* $\rightarrow$ *next*). If this generic synchronization was part of a work task with actions before and/or after it, then either the edge from *previous* must enter the middle of the task to correspond to the synchronization or the edge to *next* must exit from the middle of the task. Yet tasks should be treated like basic blocks, such that edges should only enter at the start of a task and exit at the end of a task. That being the case, then synchronizations become degenerate work tasks only containing the synchronization itself. Furthermore with synchronizations being discrete tasks representing a variety of potential coordination between tasks, the special case of locks is not treated separately by restricting its edges to the release-acquire pair, and instead retaining the general representation of each node having a partial order (1 or more predecessors and 1 or more successors).

A key feature of modeling synchronization is being able to measure the various costs associated with any atomic operation, for example: how expensive is the operation? Is there contention? What effect does contention have? Therefore, each sync action needs, at minimum, the start and end time annotations. If the sync is part of work, then both the task and any syncs each have start and end time annotations. And each sync in the work would have the appearance of a task: having incoming / outgoing edges and start / end times. Alternatively, some prior work treated syncs as edges. A sync edge would still have time annotations, where the edge would be defined from not two but four times: a start time and end time for the first action and a start and end time for the second action. If the actions temporally overlap, then the edge exists at a time in between each start and end. If instead, the actions do not overlap, then the edge is after the first and before the second sync actions.

A barrier is a common instance of all to all ordering between tasks. By representing the action as a single barrier task, rather than barriers per context, the edges in the graph pass through the single ordering point. This avoids creating a clique between the associated contexts.

2.7 Other Parallel Program Representations

Other parallel program representations had been developed. Some representations are formal in nature and support proving the properties of the parallel programs. Other representations are designed as a comprehensive programming interface to represent and then efficiently execute a parallel program. Contech's task graph is comparable to the former and is designed to encompass the latter. I will summarize several representations and frameworks not (currently) supported by Contech in this section.

2.7.1 Communicating Sequential Processes

Communicating Sequential Processes (CSP) provides a description of the program's potential actions. This formalism describes the actions that every process in a program will or may take in each step of its execution. The representation is therefore closer to the computation graph that is the theoretic basis for the task graph, rather than the task graph itself. However, CSP includes language to hide parts of a process's actions, so this technique could be used to restrict the actions exposed by a process to just the actions required to describe the interactions between processes.

A Contech task graph could be described using CSP; however, given that these task graphs are traces of specific executions, individually they do not contain the nondeterminism that CSP captures. CSP could be used to unify a set of traces from a program's execution, although such an effort is far beyond this dissertation. In so far as it might be done, contexts are treated as processes that interact with each other through locks and other synchronization. These interacting elements are distinct processes in CSP, whereby a lock is a state machine alternating between the acquire

and release events. After mapping more than one Contech task graph to a CSP description, further work would be required to show how the different descriptions could be merged through finding commonalities. Only by unifying a diversity of task graphs for a given parallel program would a CSP-based representation be then able to correspond to the static task graph (implicitly) within a parallel program.

2.7.2 Concurrent Collections

Concurrent Collections (CnC) [74] consist of three types to govern the execution of a parallel program: computational steps, control, and data. Similar to other runtime parallelism libraries, it is expected that programs will identify the parallel constructs dynamically. Computational steps define possible high-level functions that can be applied to data. Control tags determine whether a computation should execute when its data is available. This design choice enables programs to define the data used by computation in simpler terms, and thereby enable the runtime to better partition the program's execution based on greater prior knowledge of its possible execution. For example, an image is partitioned into windows and passed through a series of classifiers, where each classifier will either decide reject or emit the control tag for the next classifier to process the corresponding window. Data elements are defined in a write-once manner, such that a computational step will input data elements and output other elements. This permits multiple computations to consume the same data element; however, it introduces the need to identify when an element will no longer be required by a computational and can be garbage collected.

A parallelism expert separately defines the architecture- / platform- specific runtime that will appropriately map the parallel constructs onto the hardware. One example mapping would be deciding between assigning executing computation steps to different hardware contexts, based on the number of units as well as the communication costs between them.

A CnC-based program forms a directed acyclic graph that is generated dynamically from its execution, and the program (as well as its task graph) is generally constructed independent of the specific architecture. These traits are shared with Contech's task graph representation, and given these commonalities it would be a reasonable future step to extend Contech's instrumentation to support programs targeting CnC.

2.7.3 LogP

Part of analyzing parallel programs and parallel algorithms has been developing analytical models to estimate the runtime costs on different possible hardware. One classic model is LogP [35], which emphasized that data movement via the interconnection network can be a bottleneck at higher processor scalings. The model is based on four parameters: L - the latency of communication, o - the overhead or cost to send or receive a message, g - the minimum gap between consecutive messages, and P - the number of processors. The problem of modeling parallel programs analytically will be revisited in Section 6.1.1.

2.7.4 Legion

Legion is organized around logical regions [14], in which a set of objects are grouped and accessed through a set of permissions. Legion also utilizes *tasks*, which are a set of functionality that accesses regions. The runtime then manages these entities and schedules the *tasks* to preserve their ordering as well as the region permissions. In effect, a program using Legion is executing a task graph, where the dependencies are logical regions.

2.7.5 Galois

Efficient graph processing is becoming increasingly relevant as big data pervades computing. Graphs have notoriously bad locality and are often difficult to process

efficiently, as their accesses do not fit the traditional dependence graph model [98]. Graph algorithms often rely on fine-grained computations applied to all nodes or edges in the graph. In order to support these efforts, Nguyen, et al. [95] proposed a general infrastructure for writing graph domain specific languages (DSLs).

2.7.6 X10

X10 [29] is a second generation language in the space of those intrinsically supporting partitioned global address spaces (PGAS), and is built on top of the Java language and runtime. This support is centered around a *place*, which is a virtual pairing of computational and memory resources, such that the actual mapping of *places* to physical resources may change during execution. Parallel execution is supported particularly through asynchronous statements, quoting:

> [Asynchronous statements are expressed by the following:] `async (P) S` where `P` is a place expression and `S` is a statement. The statement async `S` is treated as shorthand for `async(here) S`, where `here` is a constant that stands for the place at which the activity is executing.

Other parallel programming support includes atomic blocks, which utilize `atomic` and rely on the runtime to select the best synchronization mechanism. X10 also provides guarantees that programs written using certain language features will not deadlock.

Altogether, I would expect that programs written in X10 could generate valid Contech task graphs to represent their execution. However, the instrumentation of virtual machine-based languages (such as, Java, C#, et cetera) as well as interpreted languages (such as Python, JavaScript, et cetera) is a problem beyond the scope of this dissertation. Given the current instrumentation tools available, it is possible to leverage them to collect the appropriate aspects of the parallel program's execution to generate the appropriate task graph.

CHAPTER 3

PROGRAM INSTRUMENTATION

There is a significant spectrum of program instrumentation. Each instance of instrumentation has costs from two components: the cost of the instrumentation and the frequency of the instrumentation's execution. Simple program instrumentation can answer questions such as how often was a routine executed or what was the largest value observed at a location.

Table 1 compares a variety of program analysis tools similar to Contech, which will separately be described in detail in Chapter 4. For each tool, I have provided the slowdown for the instrumentation as reported by its publication. Each tool potentially records three components: the dynamic control flow of the program, the memory accesses, and explicitly recording the parallel actions (creates, syncs, et cetera). Many tools only record partial information about the control flow such as the execution count of each basic block rather than the complete path, and are so denoted separately. These tools will be discussed in this chapter grouped by the framework each uses, rather than ordered by instrumentation features.

A major decision for any instrumentation approach is whether it should be source- or binary-level. Source-level instrumentation can provide greater details about the program, information that is lost during the compilation process. Additionally, the instrumentation can be better integrated into the program itself. Binary instrumentation, in contrast, does not require the program to be recompiled. However, it may not be able to accurately identify every detail about the program. Thus, one is faced with the choice between ease of use and richness of analysis.

Table 1: Instrumentation Tools with Similar Functionality to Contech (x - Support Present, b - Basic Block count without Full Path Information)

	Slowdown	Control Flow	Memory Accesses	Parallel Actions	Reference
Pin BBCount	2x-4x	b			[9] [108]
Harmony	1.2x	b			[70]
CAB Path Profiling	1.4x-2.2x	x			[53]
Pin Memory Trace	2x-8x		x		[9] [108]
PEBIL	7.7x		x		[80]
MACPO	1.5x-6x		x		[101]
ShadowReplica	2.7x	x	x		[65]
PiPA	5x	x	x		[135] [136]
Cilkview	2x-10x	b		x	[55]
ParaMeter	3x-200x	b		x	[75]
Peregrine	2x-35x		x	x	[34] [129]
Pin Task Graph	16x	x	x	x	[100]
ParaOPs	n/a	x	x	x	[102] [103]
Contech	1x-5x	x	x	x	Chapter 4

3.1 Reducing Instrumentation Overhead

In its simplest, program instrumentation is answering questions about the program's behavior. As the instrumentation is invoked more frequently or increases in complexity, the program's execution will be impacted by this overhead. Analysis tools effectively have three approaches to managing the overhead: optimizing the cost of each instrumentation event, reducing the quality or accuracy by removing specific instrumentation events, and reducing the frequency of all instrumentation events. For example, PiPA [135] is an instrumentation system focused on reducing the cost of the instrumentation events. In contrast, reducing the number of captured events is part of the approach of ShadowReplica [65], whereby it statically elides the instrumentation components that are identified as not required for the requested analysis. And PEBIL [80] is designed to switch in and out of executing instrumented code to reduce the cost of the instrumentation. Each of these instrumentation systems has taken a different approach to minimizing its overhead.

3.1.1 Inlined Instrumentation

Inlining function calls is a longstanding technique to place the called code directly inline in the calling context. This removes the overhead of the function call, as well as providing opportunities to optimize the combined code. The technique is often used as part of the initial compilation of the code and many heuristics exist to balance the benefits of inlining with the cost of increased code size. With instrumentation, the routines are often simple, yet invoked from countless locations in the program. For example, Contech's instrumentation of memory accesses can be inlined at the cost of a single instruction.

3.1.2 Sampling

One approach to mitigate instrumentation overhead is to only enable the instrumentation periodically. This approach can always improve (i.e. reduce) the overhead of existing instrumentation designs; however, this ease can result in some tool designs relying on sampling rather than a better instrumentation implementation. This design weakness can still be apparent if the instrumentation imposes some overhead when it is disabled, such as checks for enablement. Along with any periodicity of the instrumentation, the overall approach must incorporate appropriate statistical modeling to ensure that the actual program behavior is still being captured.

By minimizing the overhead, sampled tools can be used with production code, where the analysis can be running continuously to monitor for potential problems.

Arnold, et al. [6] studied periodic instrumentation and published two observations. If the instrumentation is enabled on a regular period, it may coincide with other periodic behavior in the program. Furthermore, if sampling only applies to a subset of the instrumentation, then care must also be taken to ensure that the non-instrumented regions do not count against the sampling period.

3.1.2.1 Perf / Oprofile

Rather than sampling complex instrumentation, a tool can also take samples of the current executing code in a program or running system (thus in the extreme, the sampled instrumentation is a record of every PC value of the program). By taking many profiles of a steady state, the tool can reconstruct the total amount of time spent in each region of code. This is a valuable approach to quickly observing a minimally impacted behavior of a program.

3.1.2.2 SHIM

SHIM [131] provides a very low overhead sampling of various performance counters. This tool relies on two pieces of functionality: first, it is placed within a Java VM so that targeted software instrumentation can be added to the software, and second, the performance counters on an SMT system are unified so that any hardware context can access the counters for the core. SHIM introduces a second thread to each executing application thread that consistently polls the hardware counters for that core, as well as injected software counters. This allows the system to provide sampling rates in the 10s of cycles at around 60% overhead and down to 2% overhead at 1500 cycle rate. This approach is reliant on the free resources of another core and the ability to schedule the observer thread on this context.

3.1.3 Mirroring Program

In many cases, the instrumentation required can result in the application executing orders of magnitude slower. Given the additional resources of multi-processor computers, it is possible to execute the instrumented and uninstrumented program in parallel. The design put forward by two simultaneous publications [91] [124] was to fork the program into two or more copies. The original program continues executing without instrumentation, while the copies are instrumented and run slower.

Shadow Profiling [91] opts to fork periodically in order to provide samples of the original program without excessive impact. In contrast, SuperPin [124] repeatedly forks such that multiple slices of the program are executing the instrumentation, which provides results similar to executing the entire program under the instrumentation. In both cases, the authors note that there is limited support for forking multithreaded programs and the forked copies only retain a single thread from the original execution.

3.2 Source Instrumentation

To inject instrumentation into a program, one approach is to start with the source code and extend it with the appropriate routines. Generally, the compiler is required to be modified or extended to support the instrumentation. By instrumenting at the source code level, there is greater static knowledge available about the code, such as the actual types of variables and the program's control flow graph.

As the Contech front end relies on source-level instrumentation, I will describe several tools that do the same and establish how they influenced the design for Contech.

3.2.1 GProf

GProf [49] is an instrumentation tool integrated into the *gcc* compiler framework. It works by instrumenting function calls to record execution counts and path information. Path information is collected by recording the incoming arc when the instrumentation for a routine is triggered. It determines this arc by finding the return address, thereby identifying the caller routine. Primarily, GProf is used to identify the hot code in the program, and by incorporating path information it can also provide the context of the code and frequency of its invocation. A separate study found that this simple instrumentation incurs an average of 115% overhead on SPEC 2000 Int [45].

3.2.2 ICECAP

Within the Microsoft Windows kernel, there is a function call instrumentation known as ICECAP [123]. This instrumentation records function calls and returns, along with spin locks, into per-processor buffers. Each record has an associated time stamp. As the buffers are per-processor, each record also has information to attribute which process / thread was executing given that interrupts and context switches may occur at any time. And as this is kernel instrumentation, the interrupts and context switch code will also be logged in the buffers. Furthermore, being source level instrumentation, it can be enabled and disabled on a per-source file basis. This allows the user (a kernel developer) to budget how the costs are spent: space required to record the events and slowdowns of the instrumented code.

When kernel components are recompiled with the instrumentation, they have inlined calls to the instrumentation routine. The first step of the routine is to check whether logging is enabled, thus most of the overhead is avoided until required. When in use, the kernel sets aside a region of memory that is split by processor. As it runs, each record advances the corresponding per-processor buffer pointer until the buffer is full (and consequently each processor in the trace may end at different times). Therefore, the instrumentation routine also checks whether there is space and then atomically (remember interrupts and context switches) reserves that space for the record.

After the instrumentation run finishes, a separate tool processes the buffers and reconstructs the functions involved using the debugging symbols. This output can be analyzed by other tools to determine lock usage, particularly the high contention / long critical sections; as well as understanding the common paths through the kernel. ICECAP complemented other approaches by being able to distinguish between whether expensive routines are being frequently invoked or doing significant work per call. ICECAP could also support logging other performance counters to relate

architectural phenomena, such as cache misses or branch mispredicts, with the specific locations in the code. However, most of the analysis has been superseded by ETW-based tools [88].

3.2.3 LLVM-Based Approaches

LLVM [79] is a compiler framework that provides a well-documented API for writing additional compiler passes. This lends the framework toward supporting compiler-based instrumentation.

3.2.3.1 Harmony

Harmony [70] which relies on per-thread instrumentation added by LLVM to gather basic block execution counts across a parallel program and keeps track of the thread count for each block. It instruments specific function calls to detect when application threads are not active, such as making I/O requests or waiting on synchronization. Each thread is allocated a separate thread-local buffer with one entry for each identified basic block. The tool is integrated into the compiler[1], such that at link-time it can instrument all of the provided code. In contrast, Contech has not been integrated into the compiler, and instead required additional engineering to pass and retain the instrumentation details (see Section 4.12).

3.2.3.2 MACPO

MACPO [101] uses LLVM to record a memory access trace for a targeted portion of a program. Before instrumentation, they run a separate analysis tool on the program to identify the most expensive functions in the program. Using this information, the LLVM-based analysis is then applied to the memory accesses in that function. The instrumentation is further constrained to only measure accesses explicitly made to data structures, arrays, and unions. Each access is tagged with the ID of the

[1]This is not a well documented nor supported solution.

generating core and is combined into a single log file. However, the initial performance was considered bad enough that they added support to only sample the accesses.

3.2.3.3 Peregrine

Peregrine [34] uses LLVM to record a synchronization trace for creating a deterministic schedule of the program. First, it replaces all of the synchronization operations with Peregrine wrappers. Then the program is run through the LLVM interpreter, which provides the trace of instructions and memory locations. Additional experiments showed that the overhead could be reduced to between 2x to 35x with a different approach to recording the loads / stores. The trace of the program is treated as having a total order between all of the synchronization operations, rather than the program's partial order that exists between specific operations. For example, Peregrine will order different mutexes with each other. By doing so, the tool is able to reduce the scope of possible data races. Since the final schedules will be enforced by the framework, this step can be viewed as an initial optimization on the original partial order.

Wu, et al. [129] used Peregrine to specialize synchronization schedules. The system would collect a set of executions for the program and then establish a specialized schedules. These schedules can then be used with static and dynamic analyses, such as alias analysis and detecting data races.

3.3 Instrumentation Frameworks

Collecting an arbitrary task graph requires a diverse instrumentation framework, due to the extensive data stored in the task graph. Most published instrumentation is confined to collecting the minimum required to solve the corresponding problem, and only a few tools exist that are general enough. These frameworks that could be used to record an appropriate trace for a task graph are discussed in this section. These frameworks are generally designed to instrument binaries. In the end, I elected to

develop my own framework based on LLVM's support and it will be discussed in Chapter 4.

3.3.1 Pin

Pin [85] is a well known framework designed for highly customized data collection, typically toward analyzing one aspect of a program. The original Pin included limited support for ARM-ISA; however, this support has not been maintained. The framework is effectively a virtual machine that implements the identical architecture with builtin instrumentation hooks. This requires a code cache, JIT compiler, linking traces, et cetera. One of the key pieces of the original design was optimizing how the instrumentation can impact the `eflags` register, which tracks condition codes. As condition codes are set by most instructions, the register must be preserved across the instrumentation routines; however, the only x86 instructions to preserve the flags allow them to be transfered to / from the stack. Thus Pin seeks instrumentation points that are immediately before a program instruction will overwrite the register.

After the initial design, most improvements to Pin have focused on moving the cost of the analysis to separate threads and processes. Offloading analysis to separate processes was discussed in Section 3.1.3. In order to analyze the results from the instrumentation in separate threads, the instrumentation must have some mechanism for communication. Initially, individual items could be passed between the instrumentation and analysis threads; however, there is significant overhead in doing so. Pin's Fast Buffering APIs [117] were introduced to allow support for recording the instrumented events in per-thread buffers. This support requires the events to be of a single type, although `union` or inheritance could be used to store a diversity of data (e.g., basic block IDs and memory addresses). Ansaloni and Binder [5] explored the additional support to better enable instrumentation events to queued into buffers and

later processed by other threads. This system added an adaptive model of processing that dispatches queued buffers when spare resources are available.

Cache-friendly Asymmetric Buffering (CAB) [53] worked on improving the communication of thread local buffers between instrumentation and analysis. They modeled the cost of instrumentation and analysis and concluded that the key component to improve is the communication, practically as the other costs were considered as fixed (i.e., the instrumentation and analysis was immutable). The design used a fixed-size ring buffer that is divided into communication buffers. The components will block if there are no buffers available, which prevents the system from adapting to unbalanced demand, although they recognize this limitation.

3.3.1.1 Shadow Replica

Shadow Replica [65] demonstrates the performance potential of a Pin-based tool, when using the latest published enhancements. The tool is actually two instrumentation components: an offline profiler and the actual instrumentation. A program is first passed through the offline profiler with sample inputs. The profiler finds the basic blocks in the program and determines the properties of each. It will then compute the most efficient instrumentation scheme for each basic block. For example, it may determine that certain memory accesses will not be used with the analysis program and that other accesses are based on prior addresses and do not need to be recorded. The tool also identifies fall through branches, and omits the basic block ID for the subsequent blocks. Finally, the profiler outputs an analysis program that is customized to the instrumented program, which is aware of the accesses and basic blocks that were not recorded. In the published experiments, the first pass took between 57 seconds and over 2 hours, with an average of 13 minutes. Thus incurring a significant cost to optimize the instrumentation.

In the second pass, the program is run with the final instrumentation, which records the memory accesses of note as well as the control flow. This information is enqueued into a ring-buffer shared between the instrumented program thread and the separate analysis thread. The tool further takes advantage of the fact that parts of the address space are reserved for the kernel's use. These addresses can then be repurposed to store either basic block IDs or control commands, which eliminates headers for each value. The optimizations enable the tool to store the dynamic information necessary for the analysis in 10.3 bytes per basic block.

3.3.1.2 ParaOps

Paraops combines Pin and a runtime library to collect a comprehensive trace [102] [103]. Their approach relies on Pin to collect the combined control flow and memory accesses. The runtime library flags each parallel action and stores it separately, which enables these actions to be simulated in aggregate. The focus of the design is to support a diversity of architectural simulations and do so more efficiently than existing trace-driven approaches. Given this focus, it was co-designed with the simulator and performance measurements for the instrumented program have not been published.

3.3.2 Valgrind

Valgrind [94] is a heavy-weight instrumentation platform intended to provide detailed access to the execution of program binaries. On this platform, common plugins can run that will identify leaked memory, poorly cached memory locations, and data races. Valgrind uses the disassemble and resynthesize (D&R) approach to execute the guest program. This contrasts with Pin and other tools that use copy-and-annotate (C&A) to only target the specific component of the program that is of interest. D&R is heavier-weight and requires more implementation work to support; however, it provides greater ability to express the components of interest to the instrumentation or tool.

As Valgrind executes the program, it is reading in assembly from the binary and converting it to its internal IR. This IR is passed to the loaded tool that can then apply the appropriate changes to the code. This instrumented IR is then resynthesized into native assembly.

Without any instrumentation, Valgrind runs at 4.3x slowdown; however, Valgrind is targeted at particular usage cases that are more complex than those handled by prior tools. As such, it covers a different instrumentation capability space than the lower overhead tools. One such tool, Tareador which was discussed in Chapter 2.1, uses Valgrind to generate a task graph.

3.3.3 DynamoRIO and PiPA

DynamoRIO's [23] [24] initial design was only optimizing the instructions in a program. As the framework is able to read and process the instructions from a binary, it is then valuable to be able to inspect and then introduce new instructions into the original program's stream. It was subsequently extended with an API for instrumenting the programs.

PiPA [135] uses DynamoRIO to collect a control-flow and memory operation trace; however, it requires additional software threads to process the trace, to quote "PiPA essentially obtained good performance at the expense of additional resources."[136] Additionally, PiPA has limited support for parallel threads and no support for any operations between threads (syncs, forks, joins, et cetera). The expectation for PiPA is that a user will completely process the trace while the program is running, for example the paper demonstrates modeling a cache using the memory requests from the running program. In the online analysis scenario, the analysis requires additional resources and therefore devoting further resources to the instrumentation is reasonable.

Two optimizations in PiPA's design are worth highlighting. First, rather than storing a list of memory accesses, PiPA analyzes each basic block and determines which registers are used in address calculation. Only these registers are stored. If registers are reused in a given basic block (e.g., pushing / popping function arguments), then the size of the execution profile is reduced. However, the trace processing has to reconstruct the address calculations. Second, as DynamoRIO converts hot sequences of basic blocks into larger traces, these traces can be instrumented in aggregate, which eliminates checks for overflowing the instrumentation buffer.

3.3.4 PEBIL

PEBIL [80] is built on top of Dyninst and was designed to allow efficient switching between instrumented and uninstrumented code in order to support sampling. While the full instrumentation is less efficient than Pin, reducing the example instrumentation of memory accesses to a 10% sampling rate reduces the program slowdown from 7.7x to 2.9x for NAS workloads, where most were executed with input size B. Contech's instrumentation operates at 3.85x slowdown on its subset of NAS and input class A, and against this subset of NAS only slows down by 2.6x with PEBIL's sampling.

3.3.5 QEMU

QEMU [16] provides a full system simulator, such that the application as well as the operating system can be executed. In executing both, there is additional support to generate traces of instructions executed, primarily in the native assembly of the application.

PANDA [39] is a project that provides replay and record support to QEMU with LLVM IR traces, with the goal of enabling reverse engineering efforts. One initial analysis located decryption routines in software, via both tracing backward from the

output of a failed decrypt as well as using taint analysis to trace the input's use in the program.

DBILL [86] uses QEMU to generate architecture independent instruction sequences that are translated to LLVM IR for simple binary instrumentation. This permits cross-ISA instrumentation, for example running Android apps on an x86 desktop. Without any instrumentation, this system has a 3.4x slowdown running x86 programs. A simple address sanitizer increases the slowdown to 5.7x for SPECInt benchmarks.

CHAPTER 4

CONTECH INSTRUMENTATION

Contech's task graph representation discussed previously can encompass significant program diversity, while providing a rich data set for program analysis, and the high performance instrumentation to generate a task graph will be described in this chapter. The Contech framework is composed of two parts: the generation of a dynamic task graph and the analysis of task graphs. The front end and middle layer work together to generate the task graph from a program (either source code or binary). Then back ends implement analyses and transformations that can be executed on this representation. The focus of this design is to allow easy analysis of the Contech task graph representation without a need for back end programmers to understand the details of how a task graph is generated. Each component toward generating a task graph will be explained in further detail in this chapter along with the design decisions and trade-offs, while the analysis of task graphs will be discussed in the next chapter.

This chapter is an extended version of the material that previously appeared in ACM Transactions on Architecture and Code Optimization (TACO) [100]. Figures 15, 17, and 19 all retain results taken from the version of Contech developed for publication in TACO, and are shown to demonstrate the performance improvements made since that time. Excepting where otherwise noted, the results in this chapter used the PARSEC [20] and NAS [66] benchmark suites.

4.1 Overall Design Goals

Instrumentation should be transparent to the target program, otherwise the program's behavior is affected by the instrumentation and the results can be inaccurate.

Bruening, et al. [25] put forward three categories of transparency that are desirable in program instrumentation: code, data, and concurrency. For each category, the instrumentation has three guidelines:

1. Ideally, the application should be unchanged.

2. If the application must be changed, it should be done in such a way as to appear as if it is unmodified.

3. The instrumentation should make as few assumptions as possible about the system (language, compiler, runtime, architecture, et cetera).

They noted that achieving transparency in concurrency requires particular care so as to not introduce deadlocks, or data races.

Contech's design is also predicated on two assumptions: the first is that the program is fundamentally correct and the second is that the programmer is competent / professional / et cetera. In the first, the tool does not need to protect its instrumentation from buffer overflows or other data corruptions. The program will essentially follow the semantics of the instrumented library routines for their intended purposes. In the second, the tool assumes that the program uses well established practices and avoids certain optimization techniques, such as inline assembly, manually managing large memory, or self-modifying code. This is similar to how the compiler can optimize against undefined behaviors. However, system libraries and operating system code are two places where this assumption may be violated.

Given that the tool is compiler based, Contech cannot presently instrument dynamically generated code. The primary LLVM-based approach could be combined with a dynamic instrumentation system. Or this use case would be solely served by an equivalent dynamic instrumentation tool. This choice would impose greater overheads and slowdowns from both the dynamic instrumentation performing worse

Table 2: System Configuration for Contech Measurements

Intel Xeon X5670	
# of Processors	2
Cores per Processor	6
Hyperthreading	2-way
Clock Speed	2.93 GHz
Last Level Cache Size	12MB
Main Memory Size	48 GB

than Contech (see Table 1) and the tools themselves have significant slowdown in this scenario [54], although the application is already relying on dynamic code.

The performance results presented in this chapter rely on a cluster of servers configured based on Table 2, and are primarily collected using the benchmarks from the PARSEC and NAS suites.

4.2 Front end

The first component is the Contech front end, which consists of two parts: a compiler pass for LLVM [79] and a runtime library linked to every application. Primarily, I use LLVM 3.4 with Cilk or OpenMP support to instrument programs with Contech; however, Contech has been tested with LLVM 3.2 through 3.5. Contech relies on Clang (the C / C++ LLVM front end) or DragonEgg (a plugin for gcc to support Fortran and other languages) to generate the intermediate representation (IR) for the program. The compiler pass modifies IR to include the instrumentation for recording the actions of the parallel program. As Contech covers a variety of parallel libraries, the instrumentation is applied to the IR rather than modifying the runtime libraries themselves. The pass operates function-by-function and identifies whether the function should be instrumented, or treated as a black box (e.g., `malloc`, `pthread_create`, etc). Black box functions are from a defined list; and although the compiler can detect certain operations such as atomic accesses, the programmer may have user-defined versions that would need to be explicitly identified to the instrumentation pass. This

way each function only has one static representation in the task graph: either it is a sequence of basic blocks or it is an abstracted functionality such as allocating memory or creating a thread. For each basic block, the LLVM pass finds instructions that will access memory and introduces instrumentation to record the properties of the memory access: address, size, and load versus store. Given that instrumentation occurs prior to register assignment, the LLVM-based instrumentation only records the memory operations specified by the program and not those imposed by a particular architectural model due to register allocation. Information about every basic block is also copied into a simplified set of debugging information, such that analysis tools can identify the function name, source file, et cetera even when processing a basic block ID.

4.2.1 Front End Instrumentation

The design of the instrumentation is for each action of interest (computation graph actions) by the program to be recorded by a call to an instrumentation routine in the runtime library that creates a corresponding event. The instrumentation is written in C, with functions for each type of event. The instrumentation is implemented in C rather than assembly so as to make it architecture-independent. Inserting multiple function calls per basic block would have a severe performance impact, so Contech requires programs to be compiled with clang's `-flto` flag. This flag instructs clang to perform link-time optimization, which inlines the instrumentation routines into the binary. The most common instrumentation call, recording a basic block event, requires 4 inlined x86 instructions, after the optimizations discussed in Section 4.2.1.2. Figure 10 shows the steps required for recording a basic block event, as well as the optimized instrumentation inlined into the assembly code. In this example, the instrumentation stores the basic block ID, as well as updating the position in the thread local buffer. The original, uninstrumented code had two load instructions, so the

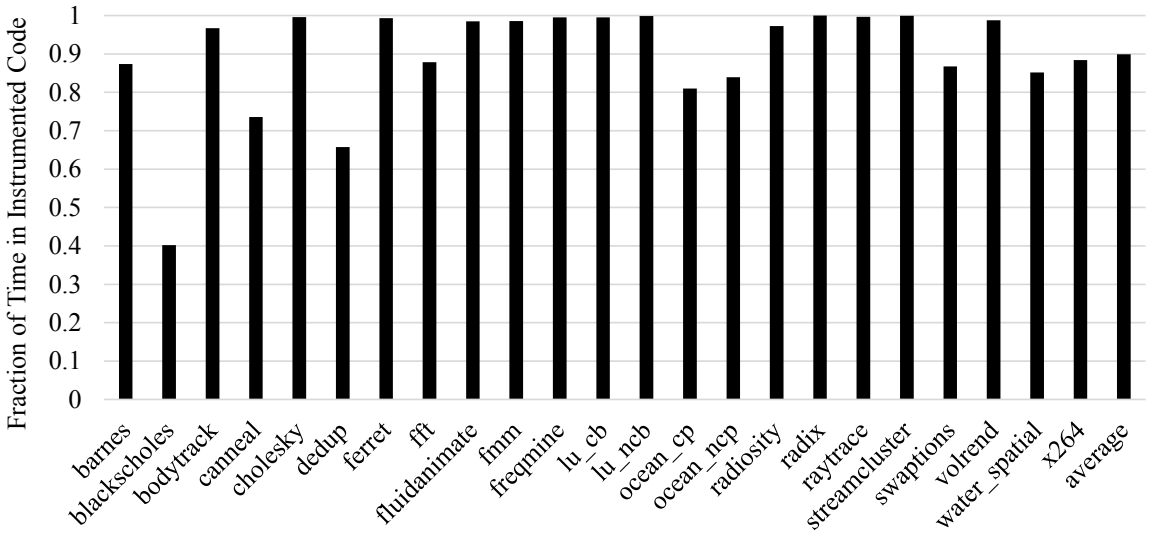


Figure 9: Code Coverage by Contech Instrumentation

instrumentation stores each address loaded by the original code. While the instrumentation introduces additional instructions, the original code has no dependencies on the instrumentation thus maintaining the existing critical path and thereby enabling the instrumentation to potentially fit into gaps where the program is not fully utilizing the core's instruction level parallelism (ILP).

4.2.1.1 Code Coverage

As Contech only instruments the source code that is passed to the compiler, a certain fraction of execution will be in other, opaque code. A portion of the execution has been stored at a semantically higher level, for example, functions such as `malloc` or `pthread_mutex_lock`; and therefore the execution of these routines is still considered "instrumented". Other library routines are not captured at any level and therefore could escape Contech's notice. In order to gain an estimate of the fraction of time in instrumented execution, Contech's instrumentation was modified to record a cycle count timestamp in every basic block. Even cycle counts can be inaccurate at this fine level of detail; however, the only time measurement of interest is when the basic blocks are separated by an uninstrumented function call.

Figure 9 presents the measurement of the time spent in instrumented execution. Specifically, the time in work tasks that is or is not in instrumented code. Functionality that corresponds to separate tasks, such as locks or thread creation is not counted either way. *Blackscholes* spends most of its time in the math library routines, `pow`, `sqrt`, `log` and `exp`. As this measurement included all time (and not just the region of interest), the uninstrumented time also includes I/O calls such as `printf` and `scanf`. Overall, there is almost no correlation between the fraction of instrumented execution and the runtime slowdown, as over half of the tested programs spend greater than 97% of the work time in instrumented code.

4.2.1.2 Reducing Event Size

The baseline implementation recorded each basic block event as a 4-tuple of 4-byte fields: context ID, type of event, basic block ID, and number of memory accesses in the basic block. Given how fundamental basic block events are to execution, these values constitute a significant fraction of the event list and even small improvements can have significant impact on the event queue size.

All events in a queued buffer originated from the same context, so each event is carrying redundant information. When the background thread processes a buffer, it adds an additional event to the stream that indicates the next set of bytes all originate from the same context. When the event list is deserialized back into events, the library recognizes these special buffer events and reconstitute the context ID for each event. The special buffer events also assist in detecting whether the event list is corrupt, as each buffer event indicates when the next buffer event will be found in the trace.

Basic blocks also have a static number of memory accesses known at compile time. Each memory access in a basic block is always of the same type and size. This static information only needs to be stored once for the program and not for each dynamic

instance. The compiler pass records this information about each basic block and at link time, this information is added to the binary so that it can be prepended to the event list, such that each memory access no longer needs to explicitly store its type and size information. When the parallel program is instrumented for 32-bit architectures and 64-bit Intel architectures, memory addresses are stored as 48 bits, which is at or above the current limit for virtual addresses [62] [77].

For practical purposes, 32 bits each is excessive for the type of event and basic block ID. These two values are combined together into a single 32-bit value. All together, these improvements reduce the space for each basic block event by 75%, when compared to the naïve approach.

The initial method of combining the type and basic block ID utilized 1 byte set to 0 as indicating a basic block. Further extracting unnecessary bits, the type *basic block* is treated as all 1 byte values which have the highest bit set to 0. All other event types then start at 128 (i.e., the most significant bit set to 1). This further reduces the size of basic blocks by approximately 6% and thereby lowers the average program slowdown by a proportionate amount. These changes limit the number of basic blocks to 8 million (2^{23}), although further engineering could provide an extending basic block event type that provides a larger field for ID values albeit with additional overhead.

4.2.1.3 Buffer Position Value

To simplify the instrumentation design, recording and serializing a basic block event consists of the basic block header followed by separate writes for each memory access. Each is a separate function invocation that is inlined. In order to minimize the memory accesses, the functions are written to pass the buffer pointer between the instrumentation calls in each basic block, which is non-obvious for the compiler to identify. This pointer is also passed to the buffer overflow checks, thus requiring the

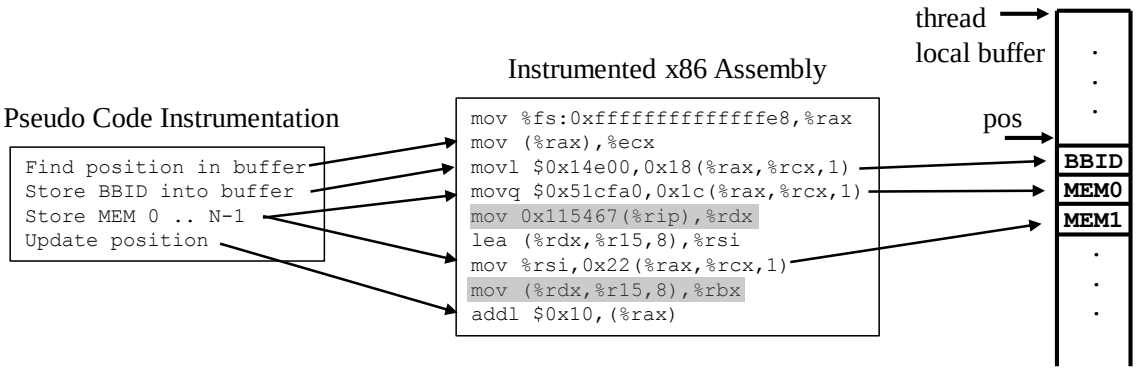


Figure 10: Runtime Instrumentation Design (grey instructions are original code)

current buffer position to be computed only once per basic block. This operation can be seen by the reuse of register `%rax` in Figure 10.

Unexpectedly, when the basic block ends with a buffer overflow check, the compiler combines the update of the buffer position with the buffer overflow check, yet it no longer reuses the initial buffer position computation. To also address this case, the instrumentation must be further updated to compute the thread local buffer pointer and the position at the start of the instrumentation block. This reduces the instructions and memory operations for the instrumentation, and has a small reduction in workload slowdown.

4.2.1.4 Instrumentation Overoptimization

The compiler is not aware that while a basic block is instrumented by a sequence of inlined function calls, in practice these calls are a logical group and cannot be moved nor reordered. In some circumstances, the compiler was moving instrumentation functions (particularly after being inlined), such that the initial basic block header was split from the rest of the basic block instrumentation. As a consequence, events would interleave and become corrupted. This required adding `fence acquire` and `fence release` instructions around the group of basic block instrumentation, which prevents the compiler from splitting up the instrumentation calls.

4.2.2 Event Ordering

A program that has been instrumented with the front-end will output an event list when it runs. Events from the same thread are stored in the event list in program order. Events generated by running threads are placed into thread-local buffers, which have a default size of 1MB. When a buffer is full, the buffer is queued into a global list of buffers. Being a global list of buffers, this gives an order to events from different buffers, which can then be used to order specific events between threads. Each type of event (create, join, sync, or barrier) relies on the global list to capture the order information, such as creator versus created or lock acquisition order, such that when an event with an ordering constraint is generated, the current buffer is queued to the global list before the thread continues.

Syncs are the most common complex event. Whether the sync is an atomic instruction, condition variable, OpenMP `single` directive, et cetera, the sync is identified either by the IR instruction or the name of the function invoked. The synchronization action is recorded with three components: the address of the action, the order of the action with respect to other synchronization actions on this address, and timestamps from before / after the action. Where possible, the synchronization action is also tagged with a type, such that it is possible to distinguish between atomic instructions, condition variables, et cetera.

The runtime framework ensures that synchronizing events are placed in the list according to their partial ordering. For example, the order of two acquire events for the same lock will reflect the execution order of the actual lock acquisitions, yet an acquire by a different thread for a different lock will not necessarily appear earlier in the event list even if it was globally acquired first. Effectively, the event list reflects the partially-ordered nature of the parallel program. Furthermore, the interleaving of critical sections observed during an execution of the program is assumed to be fixed. In a single task graph, it is not possible to determine whether certain processors could

have legally entered critical sections in a different order. To preserve the order, some events have an ordering requirement with other events and preserving this ordering requires globally queuing the event early.

4.2.2.1 Ticket Ordering

Even with the reduction in buffer size, the performance overhead in synchronization-heavy workloads is still significant. Each synchronization event (e.g., program's mutex) forces the current buffer to be queued, and the queue is protected by an internal lock in the Contech runtime. Thus independent locks in a parallel program are made dependent and their accesses serialize. The middle layer relies on the order of the sync events in the event list to order the sync tasks in the graph.

The global lock provides an ordering of events that could also be reconstituted in the middle layer. Each synchronization event is annotated with a global ordering number (a.k.a. a *ticket*), and the synchronization events are changed to no longer require a buffer flush for ordering purposes. Thus when a synchronization event occurs, it requests the next *ticket* and execution continues. The middle layer sorts the sync events corresponding to each address using the *tickets* to provide an ordering, so that it can create the appropriate dependencies between sync tasks in the task graph. This ordering constraint technically provides a global order to all synchronization events (e.g., lock acquires of different locks have an order); however, this order does not represent program behavior. A further extension to this improvement would be to use separate ticket lists for each lock in the program. Rerun [57] took a similar approach by adding sequential "timestamps" when ordering was required between otherwise independent instruction streams.

Adding tickets on sync events resulted in barrier events being received out of order, as the system would delay processing barrier events while it sought the next sync ticket. This required extending the ticket system to barriers. Originally, a

barrier event (enter / exit) would be split by queuing the buffer in between and in so doing the enter events from all threads would be seen before any exit. With barrier tickets, the queue buffer call (and associated overheads) are no longer required for each barrier, which results in a performance improvement.

Removing the queue buffer calls from condition wait improved performance, even though the queue buffer overhead averages 20% for *lu* and *ocean* workloads. As the improvement to the workloads is greater than the queue buffer overhead, the gains must also be from other components. However, this change exposed a difficult scenario for processing the event lists. In *dedup*, the workload uses multiple pools of threads. One of the threads waits on a condition variable, which is not signaled for millions to billions of cycles. As a condition wait involves releasing a lock, there is a sync event with an associated ticket stored prior to the actual call to `pthread_cond_wait`. Remember that events are processed in ticket order, which therefore requires scanning through the event list until the remaining events for the thread are found. As the thread had waited for billions of cycles, many events are scanned and queued in memory. Eventually, middle layer runs out of memory and crashes.

While both barrier waits and condition waits require a ticket before the thread waits / blocks, barriers are rarely used in an oversubscribed system and therefore do not exhibit this problem in tested programs. However, as Contech's instrumentation is applied to more programs, this problem will need to be fixed. A possible solution would be to store events from each context in a separate file. A context's events can then be processed until it blocks on a ticket, and therefore only a single event per context may be queued at any time. Most workloads have context counts in the low tens, although some OpenMP workloads can have thousands to millions of contexts.

4.2.2.2 Small Buffer Copy

When a buffer is queued early, the amount of data in the buffer can be significantly less than the buffer's capacity. Furthermore, whenever a buffer is queued, a new buffer must be allocated to continue to record events in that context. With the runtime system already allocating a new buffer, if the stored data is small (see below), a buffer is allocated for this data, which is copied into the new buffer and allows the full sized buffer to be reused. This reduces the memory requirements of the instrumentation, which improves the instrumentation performance, but at the cost of additional operations to copy the events from one thread local buffer to another. The tension is that a larger small buffer copy threshold will save memory; however, the copy will be more expensive.

With ticket ordering, most of the early queue buffer invocations are not required; however, parallel programs using condition variables or OpenMP still make significant use of this optimization. For example, *volrend*, which utilizes condition variables extensively, will use this path for the majority of queued buffers.

Figure 11 shows how sensitive the runtime system is to varying values for the threshold from 32KB to 768KB (out of the possible 1MB buffer size). Most workloads rarely trigger this path, and the results partially reflect runtime variability. Seven workloads, *dedup*, *freqmine*, *lu_cb*, *lu_ncb*, *ocean_cp*, *ocean_ncp*, and *volrend*, trigger this path 20% or more of the time and this subset of the benchmarks have also been separately averaged. The minimum of both averages is at the 64KB threshold, which also corresponds to the crossover point between time to copy the data and time required to allocate a new 1MB buffer.

4.2.3 Runtime Buffer Management

Queuing and allocating new buffers accounts for approximately 3% of total execution time, which is reduced from 14% of total execution time before the barrier ticket

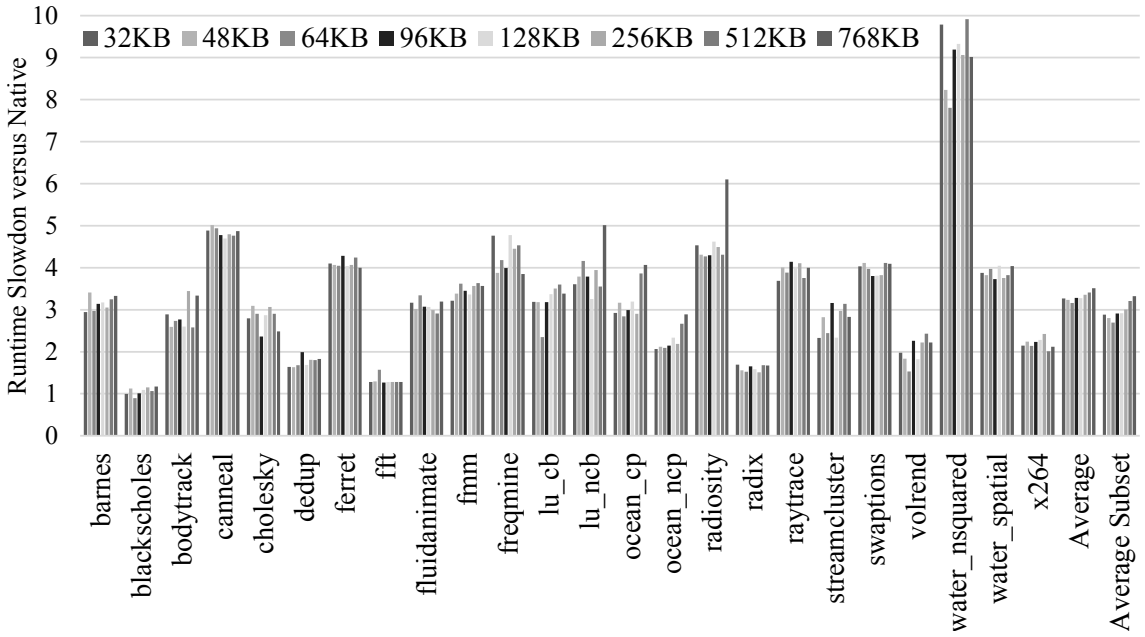


Figure 11: Instrumentation Slowdown due to Varying the Threshold for Using Small Buffer Copy Path

change. Almost all of this overhead comes from allocating the new buffer, which takes on average between 100k and 200k cycles. In contrast, the runtime can queue the current buffer in 1000 to 2000 cycles on average.

4.2.3.1 Queuing Buffer Costs

Thread local buffers are generated at approximately 6500 per second, which are queued to the single, global list. This list is protected by a standard pthread mutex. With CAB [53], they analyzed the benefits from processing buffers in parallel with the instrumentation. Their observation was that it was vital to minimize the communication overhead. While Contech's current implementation could be further improved below 1000 cycles, the important aspect for its current performance is to have a tail pointer, such that the thread local buffer can be quickly queued to the end of the list of waiting buffers. If Contech were run on a system with significantly more processors, it may be advantageous to improve the queuing system; however,

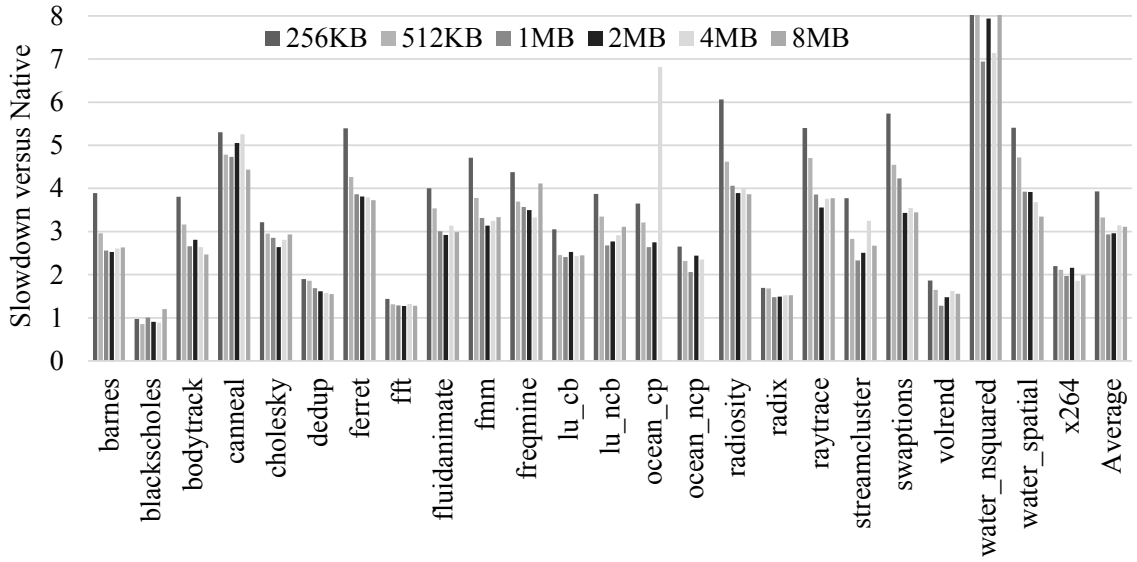


Figure 12: Impact of Changing Buffer Size on Contech Runtime Slowdown

the current performance is almost solely due to acquiring and releasing the mutex and is furthermore less than a tenth of 1% of total execution time.

4.2.3.2 Memory Allocation

Prior work has shown that instrumentation performance can be improved by modifying the size of the buffers [117]; however, this was potentially an artifact of the cost of using guard pages to detect buffer overflow. Figure 12 shows the runtime slowdowns for PARSEC workloads as the buffer size is increased from 256KB to 8MB. The results of the different buffer sizes are diverse. Above a certain threshold in size, most workloads are minimally affected by the buffer size. Several workloads are negatively impacted by larger buffers. These workloads generally use condition variables and are therefore utilizing the small buffer copy path. Furthermore, as the buffer size increases, the percentage of buffers reused decreases and the memory requirements of the instrumentation increases, and the overall memory usage of the instrumentation correlates with the parallel program's slowdown. Now, as each workload performs best with a specific buffer size, if the runtime system was able to tailor the buffer size

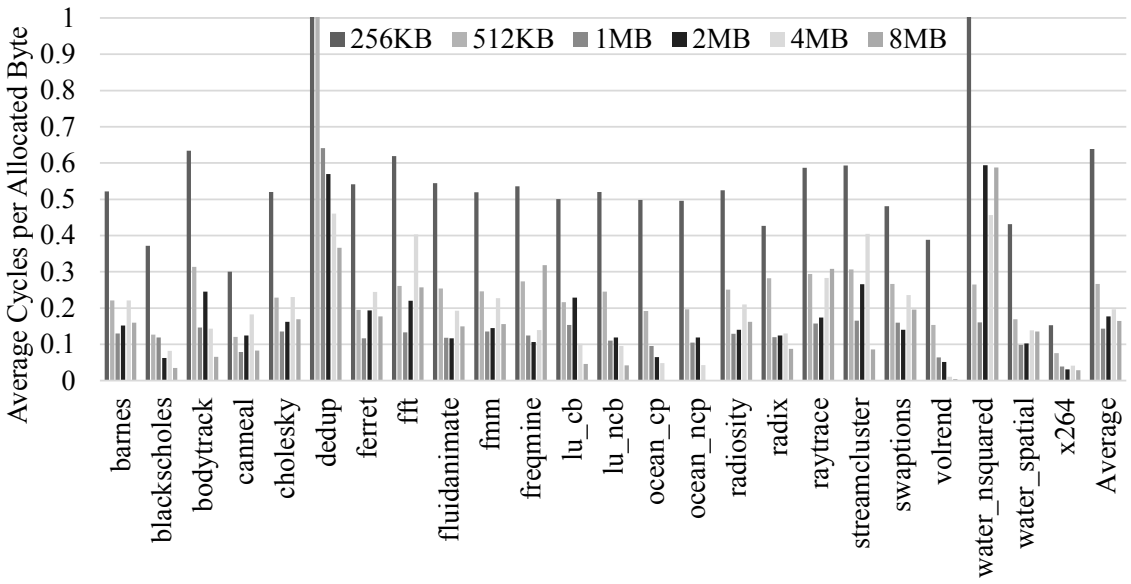


Figure 13: Cost to Allocate Buffers based on the Size of Buffer

for each program, then the average slowdown would be 2.6x versus the best single configuration average of 2.8x with 1MB buffers, excluding *water_nsquared*.[1] Figure 13 shows the average cost of allocating thread local buffers when factoring in the size of the buffer. For most workloads, the cost is declining even at 8MB and therefore there is less overhead per unit of buffer space. The overall average is lowest at 1MB, which supports it as the default buffer size. I would also note that *dedup* has particularly high allocation costs due to context switches impacting the allocating thread, and since the next thread is also making progress, the program itself is not slowed down by this high allocation cost.

Maintaining a hard limit on the largest size of buffer that can use the small buffer copy path, rather than scaling with the buffer size, restores much of the performance lost from the larger buffer size. However, this change results in wasting a significant quantity of memory. For example, *volrend's* memory usage increases by over 30% such that the memory used is 20% larger than the trace itself. Furthermore, removing the

[1] *Water_nsquared* only barely fits in the available memory without requiring tracing to be suspended (see Section 4.4.2), which limits the ability to collect timing measurements and will therefore be excluded from averages in order to enable comparison between different instrumentation setups.

queue buffer call from the condition wait path also mitigates the performance impact of the larger buffers. However, both mitigation solutions do no better than generally returning to parity with the 1MB buffer size.

Each allocated buffer is of two parts: the space for the data and the header fields. The header fields require 24B on a 64-bit system. As `malloc` passes requests for more than 128KB to `mmap`, which requires the allocation to be multiples of the page size, this results in an allocation for 1MB + 4KB. For most workloads, there is a bimodal distribution where most requests are made to `mmap` requiring roughly 150k cycles per allocation. These allocations require the kernel to assign new virtual addresses to the page table, which may require invalidating the existing translations on all processors executing this process as well as assigning zero'd pages to the new translations. The Contech runtime will also reuse buffers after the contents have been written to disk; allocating a buffer for reuse averages 2000 cycles, which is primarily the cost of acquiring and releasing the mutex protecting the list. However, on *dedup* the requirement to queue the buffer on each condition variable pushes up the number of memory allocations and the oversubscribed number of threads results in many context switches while attempting to allocate new buffers. The context switch is inferred from recording allocation costs as high as 50 million cycles made on the small buffer copy path.[2]

An important feature of Contech's buffer management is the ability to reuse the thread local buffers. After writing the contents to disk, the background thread places the buffers back into an allocation list. Given the disparity between generating data and writing it to disk, most allocation requests must go to `malloc`. To measure the benefit this feature provides, I disabled returning written buffers and instead the background thread held them until the end of execution. This result is shown in

[2]If garbage collection (GC) was present, then it would be possible that these events represent triggering the garbage collector. Or if the memory usage was higher, then these costs could represent paging as the OS prepares zeroed pages for the allocation.

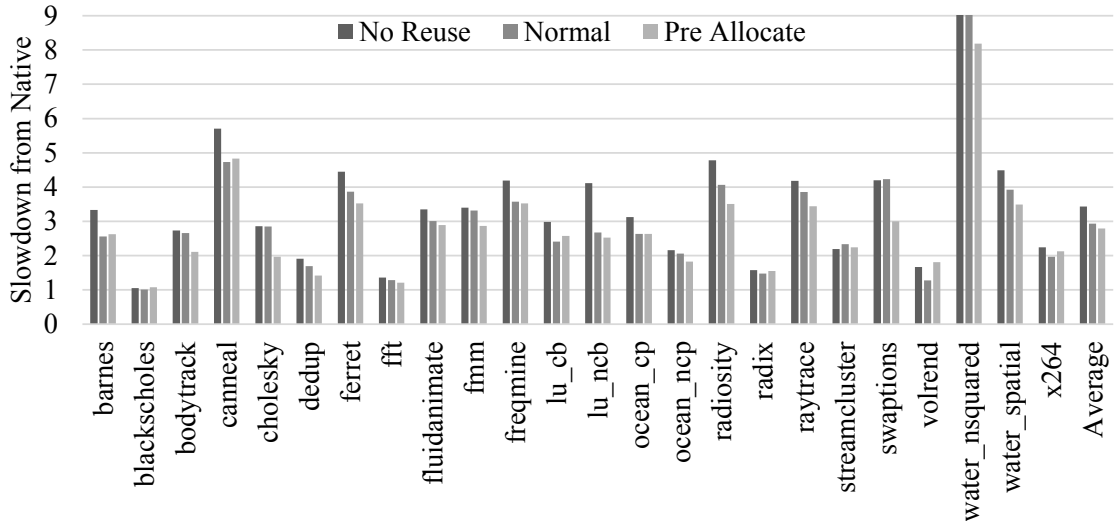


Figure 14: Varying Buffer Allocation and Reuse

Figure 14 as "No Reuse". The standard behavior of the Contech runtime system is displayed as "Normal" for comparison. The third set of bars is for preallocating the thread local buffers. For simplicity of design and testing, the system preallocated 10,000 buffers before beginning execution. More than two thirds of the benchmarks can then satisfy all of their allocations from the reuse allocation list. As a consequence, many workloads benefit from preallocating the buffers. The main drawback of this approach is that this still takes time, which delays the process from launching the program to generating a task graph, even if the instrumented core executes faster. In the future, setting a preallocation quantity may be added as a feature for workloads / analyses that are highly sensitive to timing and slowdown.

4.2.3.3 Buffer Overflow

When writing events to the buffer, the system needs to ensure that the buffer is not overflowed. As the runtime system cannot spread a basic block (or other event) across two buffers, the system can check the quantity of data added thus far before inserting a new event. Each check verifies whether a certain margin of space is still available in the buffer (default of 1KB). As each check establishes that sufficient space is still

available in the buffer for (on average) multiple basic blocks, many of the checks would be redundant and are not inserted into the code. Basic blocks average 1.7 memory operations per basic block (for PARSEC and SPLASH) and 2.8 for NAS, such that the blocks take an average of 14 and 20 bytes respectively and therefore between 50 and 70 average blocks can fit into the margin of space checked. However, there are also basic blocks with a large number of memory operations, such that a single one of these blocks is larger than the check margin and therefore these blocks always have a buffer overflow check to ensure that the event for this basic block will not exceed the margin of buffer safety. Having reduced the number of buffer overflow checks in the instrumented code, the remaining checks for buffer overflow are around 7% increased execution time. While other instrumentation approaches have proposed canary values [135] or guard pages [117], the additional gains would be minimal versus the added complexity, particularly when additional refinements to the check locations could further reduce the number in instrumented code. Even with the guard pages approach, the authors noted that given the cost, they added additional instrumentation to check for overflows and only relied on the guard pages as a fallback option.

4.2.4 Trace Formats

To recap, the trace written out by the front end consists of sequences of serialized events with special header events to indicate which context generated the next sequence of events. Given the default buffer size, each event sequence delineated by header events is close to 1MB in length. Together, the sequences of events form the event list, which can be processed to generate a task graph.

4.3 Middle Layer

The middle layer processes the event list generated by a front end to convert this trace into a Contech task graph, following Algorithm 1. Contiguous streams of basic blocks

ALGORITHM 1: Assign Events to Tasks

```
input  : Event List
output: Task Graph
for e ∈ EventList do
    if e.Type == BasicBlockAction then
        if Context[e.context_id].getCurrentTask ().getType () != e.Type then
            Context[e.context_id].makeNewTask (e.Type);
        end
        Context[e.context_id].getCurrentTask ().append (e);
    end
    else
        Context[e.context_id].makeNewTask (e.Type);
        // Find other predecessors to the new task
    end
end
```

in the event list are combined into tasks. Each work task accumulates a list of basic block IDs and memory accesses. When the middle layer encounters a synchronizing event, it closes the current task for the context and determines dependencies between other synchronizing events in the task graph. The middle layer considers the four types of dependency tasks: creates, joins, syncs, and barriers. In doing so, it is partitioning the task graph into the types of tasks, while creating the graph. This avoids any need to find the partitions after the graph has been created.

In processing the events, three events may be queued for later processing: creates, syncs, and barriers. If an event is queued, all subsequent events for that context ID are also queued. To simplify the processing of the event list and generation of the task graph, the middle layer will delay processing a child created event until that context has been created by another context, the parent. Syncs and barriers both contain ticket numbers and are treated identically, except in that the tickets themselves are separate orders respectively. As the tickets are to be observed in numeric order, if an event contains a ticket that is later, then it is queued and waits until it is its "turn" to be processed. In general, the next ticket should be in one of the next buffer sized units in the event list; however, as noted in Section 4.2.2.1, it is possible to construct

scenarios where the appropriate buffer containing the next ticket is separated by millions of events and this queuing of events may eventually exceed the available memory. The middle layer could process the events into a separate, albeit incomplete task graph that would be integrated after the appropriate ticket events are processed.

To avoid any disruption of the running program, the middle layer does not process the trace until the instrumented program has completed. As with the front end, the middle layer also utilizes a background thread to write out tasks that are identified as complete. In this component, tasks are complete when all predecessors and successors have been identified, and not necessarily when the middle layer has begun processing a successor task. As the tasks can be stored in any order on disk, tasks can be written out as soon as they are complete and not queued into a particular order, with this the middle layer effectively maintains only one work task per context actively in memory while the others are being sent to disk and then deallocated. This lowers the memory usage and improves the performance of generating a task graph.

Prior to writing out each complete task, the serialized data of the task is passed through a compression library, and by compressing tasks individually the graph representation supports efficient random accesses. Having prepared the task graph, the middle layer also records a breadth-first traversal of the task graph, starting at Task 0:0 and following the task dependencies. Given multiple possible tasks to visit next, the middle layer checks the timestamps, if present, and selects the task with the earliest timestamp for when it started. This traversal provides a common order for most analysis tools. This is also the one case where timestamps play a role in generating a task graph.

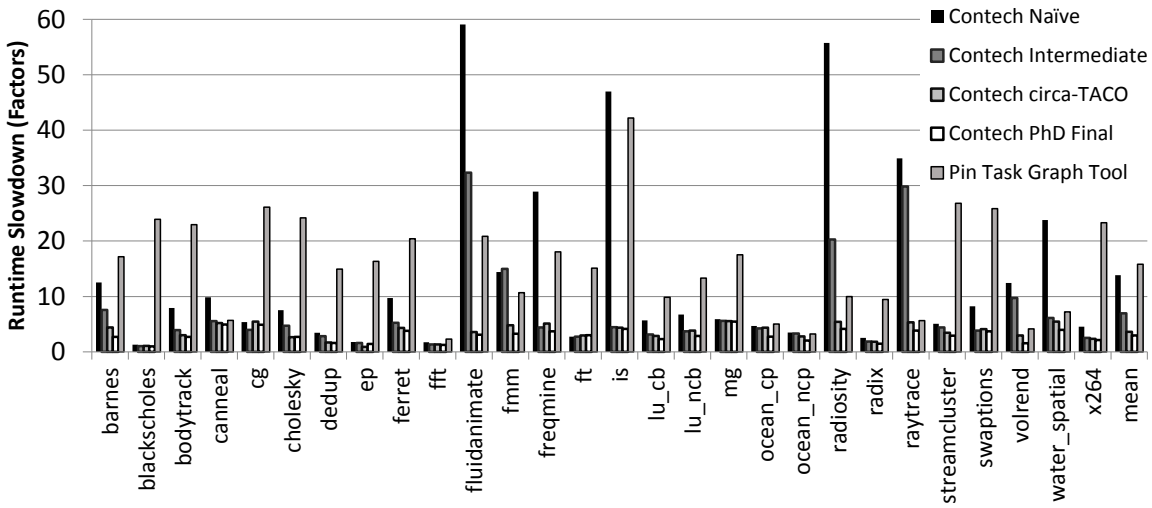


Figure 15: Comparison of Contech and Pin Runtime Slowdown for PARSEC and SPLASH (simmedium, 16 threads) and NAS (A, 16 threads)

4.4 Instrumentation Performance

This section discusses the performance measurements for the resulting instrumentation. The native baseline, Contech instrumented, and Pin-based runs of the benchmarks are compiled with the same scripts and executed within the same environments. This accounts for some of the known measurement biases [92]; however, all executions are run on real hardware and may be impacted by the following effects: the power / frequency level of the processors, the temperature of the environment, as well as operating system effects including scheduling and interrupts and even the quantity of zeroed pages available at the start of the program's execution. Given the shared usage of the machines which limits the ability to directly control these effects, the presented results are the average of 5 runs and are compared against the baseline execution time.

Figure 15 shows the performance of four Contech implementations using the features described in Section 4.2.1.2, along with a comparable Pin tool implementation

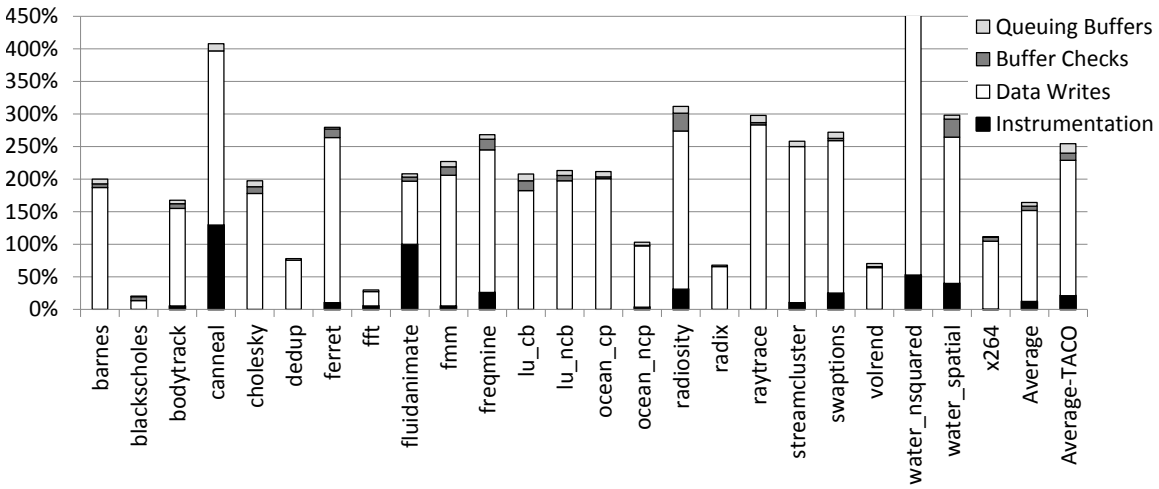


Figure 16: Overhead Increase from Runtime Instrumentation over Uninstrumented Programs

(see Section 4.6). The naïve implementation is outlined in Section 4.2. The intermediate implementation is improved by reducing the event queue size. The implementation circa-TACO includes other performance techniques, primarily synchronization tickets. The final implementation removes the extraneous forced queuing of buffers, reduces the basic block events to 1 bit, explicitly passes the thread local buffer pointers within the basic block instrumentation, and identifies redundant address calculations. The original, naïve implementation of the Contech runtime had significant overhead for a majority of the PARSEC benchmarks. Taking advantage of compiler and architectural knowledge to change the structure of events, the runtime overhead was significantly reduced; however, a small number of benchmarks, such as *fluidanimate*, *radiosity* and *raytrace*, still exhibited higher overhead than the remaining benchmarks. These benchmarks make significant use of synchronization. By changing how the synchronization is handled in the runtime, the overhead is reduced and exhibits little variability between benchmarks. The final slowdown is between 1x to 5x with an average of 2.8x, excepting *water_nsquared* with almost 7x overhead.

By selectively disabling parts of the runtime, the overhead imposed by each component of Contech's runtime is measured across all benchmarks. Figure 16 shows the

runtime increase with each instrumentation component selectively added in, all based on the current version of the Contech instrumentation.

For most workloads, the processors have sufficient resources to execute the instrumentation with low overhead (12% increase in execution time), which is reduced from 20% partially from the struct pointer change. Having removed most of the checks for overflowing the thread local buffer, the remaining checks only account for a 7% increase in the execution time and is also improved from the struct pointer change. The time required to allocate a new 1MB thread local buffer and queue the existing one into the global queue increases the execution time by around 6%, and has been improved by avoiding more cases where the buffer is queued early. Nearly all of Contech's overhead comes from the impact to the data cache from the buffer cache forcing out application data.

One exception is *fluidanimate*, which has high instrumentation overhead due to frequently performing lock and unlock operations on mutexes. Each mutex operation is recorded as a sync event and each sync requires a ticket. The ticket is a global, atomic value and therefore measurably contributes to the overhead of the instrumentation; however, acquiring a global ticket is still significantly less expensive than the original implementation that queued buffers early.

Other workloads, such as *water_spatial*, have high buffer check overheads. They have similar percentage of basic blocks with buffer overflow checks as the workloads with lower overheads. And there is almost no correlation between this overhead and the branch misprediction rate.

4.4.1 Data Write Performance

Writing the events into the thread local buffer still takes the majority of execution time even after the previous improvements. Changing these writes to use non-temporal, write-combining writes (i.e., SSE2's `movnti`), did not reduce this overhead and in fact

slowed down from 3.5x to 4.0x for the tested benchmarks. Moving the instrumentation into a single sequence (rather than interleaved) has no impact on performance, not even when `movnti` is used. With a high rate of memory traffic and regular nature of the instrumentation's memory writes, there is also little benefit to prefetching the thread local buffer.

Given the impact of filling the buffers on the cache, there is a stronger correlation between slowdown and time to write the data than the runtime itself (0.78 versus 0.49). The strongest correlation is between the slowdown and the amount of data written at 0.85. This correlation declined to 0.8 after reducing the basic block type to 1 bit, showing that as the data written is reduced, other components will increasingly contribute to the runtime overhead. With that in mind, any redundancy or unnecessary data from the instrumentation could improve performance. There are three known redundancies in the trace: path information, static addresses, and redundant address calculations.

Path information is a technique to represent sequences of basic blocks with a single identifier. Instrumentation is then added to only count when an entire path has been traversed [10]. Adding this instrumentation is more complex, particularly when the basic block instrumentation has already been tightly coupled with the memory access instrumentation. Furthermore, the basic block identifiers are currently less than 25% of the total trace size, which constrains the potential gains from using paths in lieu of basic blocks.

A series of experiments also explored static, global addresses and their impact on the instrumentation performance. First, the compiler instrumentation has been extended to identify accesses to global variables. Table 3 gives the measurements across PARSEC and NAS, where the average is 5.2% of accesses across the benchmarks are made to global variables. Dropping any load / store of the identified globals reduced

Table 3: Static Global Accesses per Benchmark

	% Accesses to Globals	Slowdown (Data Write)		% Accesses to Globals	Slowdown (Data Write)
barnes	15.5%	4.42 (3.04)	is	1.2%	4.38 (2.61)
blackscholes	13.1%	1.09 (0.08)	mg	0.0%	5.56 (4.44)
bodytrack	0.0%	3.02 (1.76)	lu_cb	1.0%	2.88 (1.66)
canneal	1.6%	5.21 (2.57)	lu_ncb	0.3%	3.85 (2.28)
cg	0.0%	5.45 (4.06)	ocean_cp	23.3%	4.35 (2.87)
cholesky	0.4%	2.67 (1.58)	ocean_ncp	23.7%	2.78 (1.62)
dedup	0.2%	1.70 (0.74)	radiosity	0.4%	5.44 (3.60)
ep	0.0%	0.92 (0.53)	radix	15.4%	1.83 (0.86)
ferret	0.0%	4.30 (3.00)	raytrace	0.1%	5.32 (4.04)
fft	1.8%	1.32 (0.27)	streamcluster	1.4%	3.47 (1.52)
fluidanimate	31.8%	3.57 (1.16)	swaptions	0.0%	4.10 (2.67)
fmm	0.3%	4.79 (3.29)	volrend	5.1%	2.95 (1.45)
freqmine	5.6%	5.10 (3.15)	water_spatial	4.1%	5.45 (3.44)
ft	0.0%	2.96 (1.77)	x264	0.0%	2.37 (1.25)

the trace length by an average of 3%; however, there was no observed change in runtime overhead.

In contrast, removing all memory operations from the trace reduces the slowdown from 3.6x to 2.1x. Given that the memory operations are approximately 75% of the trace, the overhead of data writes (220%) should be reduced by a factor of 4 to about 55%. This would give an overall reduction in overhead of 1.65x, which is close to the observed reduction of 1.5x. A slowdown of 2.1x to record path information and parallel actions is comparable or better than similar tools, see Table 1.

4.4.1.1 Redundant Address Calculations

PiPA [135] and Shadow Replica [65] demonstrated the capability of identifying when multiple memory accesses use the same or similar address calculations and emphasized this feature as part of reducing their total trace size for lower overhead. These tools, utilizing DynamoRIO and Pin respectively, relied on identifying common registers in the assembly and then finding the different offsets therein. Stack accesses

are a common example in prior work. While Contech does not capture explicit stack accesses, Contech can use this technique to reduce the number of addresses that must be stored, which results in less data generation and lower overhead. Prototype measurements showed that this change would reduce the number of memory operations stored in each basic block from 2.25 to 1.67.

The Contech front end was extended to check each memory operation for whether its address is based on another operation. Contech restricts itself to cases where the memory operations and their address computations occur in the same basic block. It is possible that other cases of memory address redundancy could be identified; however, the $O(N^2)$ naïve scan of the memory operations in the basic block increases the compilation time by approximately 5%. LLVM supports address calculation primarily through the `GetElementPtr` instruction, which takes a base pointer and 0 or more offsets that iterate through the nested types of the base type. An address calculation is redundant if Contech can compute a static difference between the offsets for the same base pointer. Contech can then store this difference in the static basic block information and recreate the address when the trace is processed into a task graph. Actual testing identified flaws in the prototype measurements, and the number of memory operations stored at instrumentation time per basic block was reduced to an average of 1.84, with a corresponding reduction in runtime overhead for PARSEC from 3.11x (after extraneous forced queuing and single bit basic block event headers) to 2.77x, as shown in Figure 15.

4.4.2 Background Writing Performance

Buffers in the global list are written out by a dedicated background thread. This design minimizes the overhead in the running threads to only that required to record an event. Should the queued event lists reach the threshold of the system's memory limit (default of 90%), the runtime will suspend further execution until all queued buffers

have been written to disk. The runtime will also insert events for each suspended thread to indicate that the thread spent time paused.

The instrumentation generates data at an average rate of 5GB/s for each benchmark (between 1.6 and 8.1GB/s). This data is being written out in the background while the program is executing; however, most hard disks do not support 5GB/s transfer rates, so the event list buffers in memory during program execution. As mentioned earlier, should the queued buffers reach the memory usage threshold, the instrumentation suspends the programs execution until there are no queued buffers. Moving this background writing to multiple threads could enable online analysis and compression [53], yet greatly increase the processing requirements of the running machine. The following section will further discuss some of the difficulties of implementing this additional background support.

4.4.2.1 Streaming Compression

The disk, using the OS file cache, consumes the trace at 100 - 900 MB/s. Rather than write the results immediately, the trace could be first compressed. Simple experiments indicate 500MB/s is an approximate bound for a simple buffer copy, when copying 512MB between two memory locations, while smaller sizes will operate closer to 1GB/s. Any compression algorithm would require other calculations in addition to the buffer copy (i.e., read the original trace and write a compressed trace).

Ibtesham, et al. studied general purpose compression tools applied to application checkpoints in high performance computing [59], and found that the compression rate was in all cases less than 100 MB/s and often less than 10 MB/s. However, the checkpoints were compressed by 50% or more. Instead of using general purpose compression, prior work has demonstrated storing a path (i.e., a sequence of basic blocks), rather than recording the events as individual basic blocks, and deciding on the description of the paths using context free grammars [78] can provide good

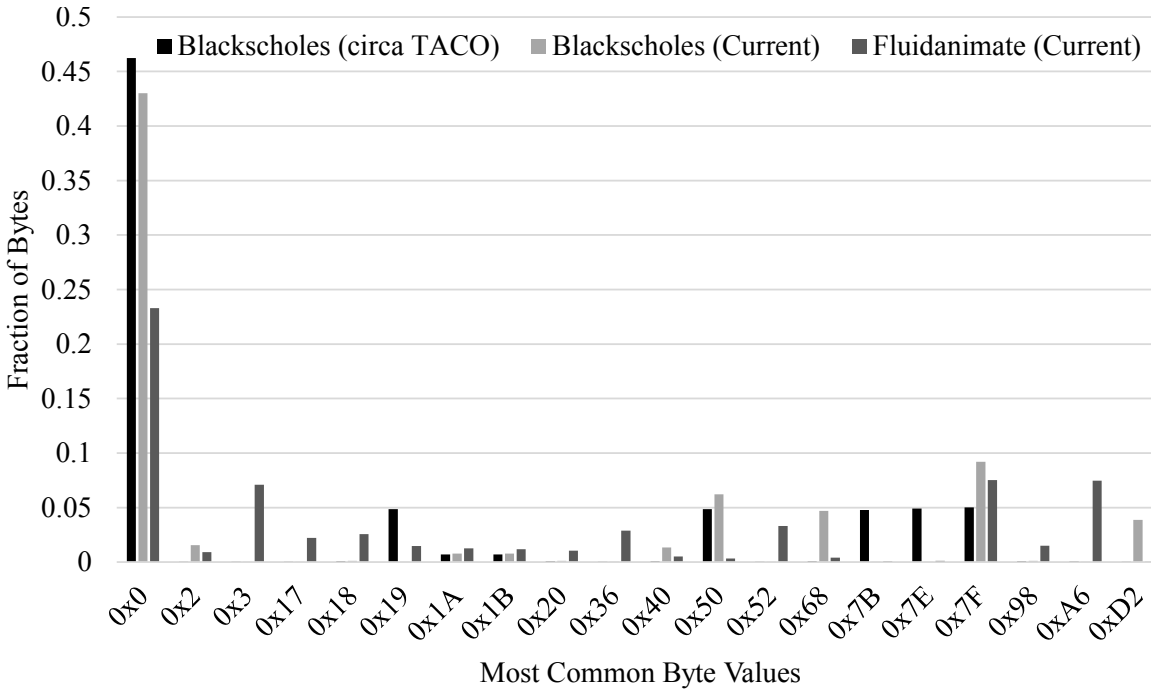


Figure 17: Frequency of Byte Values in Contech Trace

compression ratios. This trace format has been extended to include a representation of lock operations [47] and data dependencies [115]. In experiments using Contech event traces, the compression algorithm used in the prior work (Sequitur) only runs at 2 - 4 MB/s. Kanev and Cohn [71] demonstrated that tying simple predictors from a functional simulator to the trace collection can predict many instructions in a trace and thereby eliminate the need to store them. In general, these approaches are guided toward the goal of minimizing trace size rather than maximizing the trace collection rate. COP [97], in contrast, tries to efficiently compress memory blocks to provide space for storing error correcting codes and can do so with minimal impact on the program.

Studying the frequency of values from several traces, the most common value is 0, see Figure 17. This figure shows three distributions of byte values. Two are taken from running *blackscholes*, which is a small workload (few basic block IDs) and almost solely basic block events. The first bar is from the version of Contech published in

```
for (uint64_t i = 0; i < fsz; i++)
{
    v = fc[i];
    buf <<=1;
    bits++;
    if (v != 0)
    {
        buf |= 0x1;
        buf <<= 8;
        buf |= v;
        bits += 8;
    }

    if (bits > 8)
    {
        v = (buf >> (bits - 8));
        bits -= 8;

        wCount ++;
    }
}
```

Figure 18: Simple Compression Algorithm

TACO, which used a zero byte to indicate basic block events. The second bar shows how this distribution changes with the basic block event using 1 bit. The third bar comes from *fluidanimate*, which is a synchronization heavy workload. Comparing the first and second bars shows that reducing the basic block events to 1 bit has some reduction on the fraction of 0 values. Between the second and the third bars, there are commonalities between workloads, such as the common occurrence of the value `0x7F`, which come from addresses on the stack. A basic calculation showed that replacing any 0 byte with a single bit and prepending an additional bit to all non-zero values would reduce the trace size by 15 - 30%. This algorithm is implemented in Figure 18. Measurements showed that this code could process data at approximately 350 MB/s.

An initial test with *fluidanimate* demonstrated that a simple scheme would not be sufficient for any performance improvement, as the performance regressed from 3.57x to a 5.96x slowdown. In addition, the time to write the data to "disk" increased from

Table 4: System Configuration for Contech Measurements on ARM NVIDIA Tegra K1

Processor	ARM Cortex A15 r3p3
# of Processors	1
Cores per Processor	4
Clock Speed	2.3 GHz
Last Level Cache Size	2 MB
Main Memory Size	2 GB

Table 5: Runtime Overhead for PARSEC (simsmall) with Clang and Optimized Contech (without lto) on ARM

Benchmark	Run Time Slowdown
blackscholes	4.07
cholesky	31.47
dedup	6.41
volrend	11.03

5.5s to 40s, as the write speed now matched the compression speed instead of the file cache speed. There are two theories to this regression: first, this scheme introduces a high CPU usage thread to the running system that competes with the existing code, and second, the lower write rate increases the memory requirements and evicts more data from the caches. However, while a detailed analysis has not been conducted, the scheme was able to compress the trace to 83% of the original size.

It has also been suggested that the memory bandwidth could be reduced with some variation of inline delta compression. In this scheme, the runtime instrumentation would include additional operations to compute the deltas between bytes, compress the deltas and then store the compressed data into the thread local buffer. The question is whether the cost of these operations would be mitigated by the reduced memory traffic to the thread local buffer.

4.5 Running on ARM ISA

To demonstrate support for other ISAs, a subset of PARSEC benchmarks with Contech instrumentation were selected to be executed using an ARM-based system (see

Table 4). Given the small size of the system, these workloads used the *simsmall* input, with the slowdowns in Table 5, based on the Contech version from TACO. An LLVM bug prevented the link-time optimizer from working properly on ARM; the most common instrumentation calls are marked force_inline, but the instrumentation is not being optimized to the same degree as on x86 and therefore increases the slowdown. In addition, the ARM processor does not have the same quantity of spare microarchitectural resources for the instrumentation, and only generates data at less than one quarter the Intel processor's rate. The only change to the Contech framework required for ARM was adding 1 line of assembly to access the time stamp counter (PMCCNTR); however, the architecture support is less clear than on x86.

4.6 Comparison with Pin

Pin is a widely used, high performing instrumentation tool that has support for collecting similar data to Contech's task graphs. I searched prior work and found several tools that collect subsets of the data recorded by Contech (the full list is in Table 1): the overhead of recording a memory trace alone (ignoring basic blocks, as well as attributes of the memory accesses) ranged between 2x and 9x [9] [80], recording the memory trace with attributes and basic block information averaged a 16x slowdown [135], and recording the program's DAG and instruction count incurs 2x - 10x slowdown [55]. As Contech's instrumentation is more extensive than these tools, I implemented my own Pin tool that is functionally equivalent to Contech's instrumentation, targeting Pin 2.13 and using Pin's Fast Buffering APIs [117] to record basic block IDs and memory operations, with read or write and size attributes, along with events for synchronization actions. I incorporated optimizations to the Pin tool including those in Section 4.2.1.2 such that the performance would be comparable to prior work.

Running the PARSEC and NAS benchmarks using Pin, the execution time is recorded to estimate the overhead imposed by Pin. The results are the average of 5 runs, and the measurements are for the whole program execution. Figure 15 shows the runtime slowdown for the benchmarks, comparing Contech front end versions with Pin. This set of measurements show that the current implementation of Contech's front end performs better than a Pin-based implementation does on all workloads. The Pin slowdowns also show significant variation across the PARSEC benchmarks, varying from 2.3x to 42x and averaging 15.8x.

The hardware performance counters are measured using the tool *perf* to compare Contech's and Pin's instrumentation. Being a dynamic instrumentation tool, Pin significantly impacts the instruction cache performance with an L1I cache miss rate of between 2.2 - 8.0% averaging 4.4% versus Contech's range of 0.1 - 7.1% with an average of 0.7%. The uninstrumented code's instruction cache miss rates range from 0.02 to 4.0% with an average of 0.5%.

Figure 19 shows the increase in architectural events from Contech's and Pin's instrumentation versus the baseline programs. In all cases, the instrumentation has increased the operations and instructions executed by the programs. Along with the measurements in Figure 16 where the instrumentation instructions were shown to have low overhead, the runtime cost of the additional operations recorded with *perf* is primarily the increased cache pressure. As most of the architectural miss rates remain similar between Pin and Contech and in some cases are better than the native code, viewing the results by the total number of events better illustrates the slowdown from Pin.

The recent improvements to Contech have reduced the loads by 30% primarily by ensuring that the thread local buffer values are only loaded once per basic block (see Section 4.2.1.3) and the stores by 8% by removing redundant addresses from the event list (see Section 4.4.1.1). These changes have also reduced the instrumentation

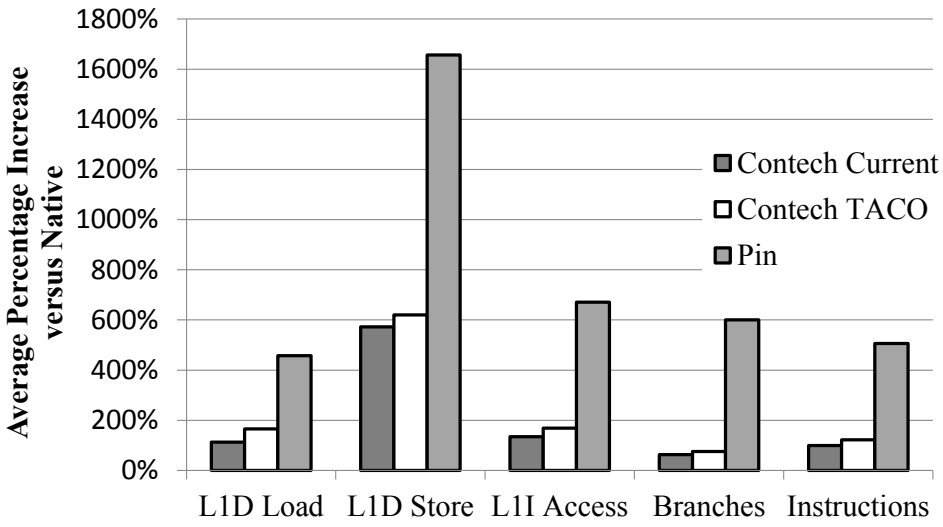


Figure 19: Average Increase of Architectural Events over Uninstrumented Execution

instructions, such that the instruction cache (L1I) accesses and dynamic instruction count have been reduced by 20% each.

4.6.1 Instruction Execution Rate

While the instrumentation approaches increase the overall load of instructions between 100% and 500%, as shown in Figure 19, the rate metric of CPI may show an improvement with the instrumentation. For example, the CPI for Splash 2 workloads is improved when running with the Pin-based instrumentation, such as *radix* having a 50% reduction in the CPI from 3.49 to 1.74.

While the number of instructions added per basic block are few, each is particularly long. For example, the common load of the thread local buffer is 9 bytes, the store of a basic block ID is 8 bytes and storing memory operations are 5 - 9 bytes in size (whether the address is in a register or an immediate). Using *fluidanimate* as an example, the average size of the instructions in the native binary is 4.8 bytes, while the added instrumentation averages 5.0 bytes. Furthermore, approximately 60 - 70% of the instructions in the binary have been added as part of the Contech instrumentation. Some of these instructions correspond to the runtime library routines, while others

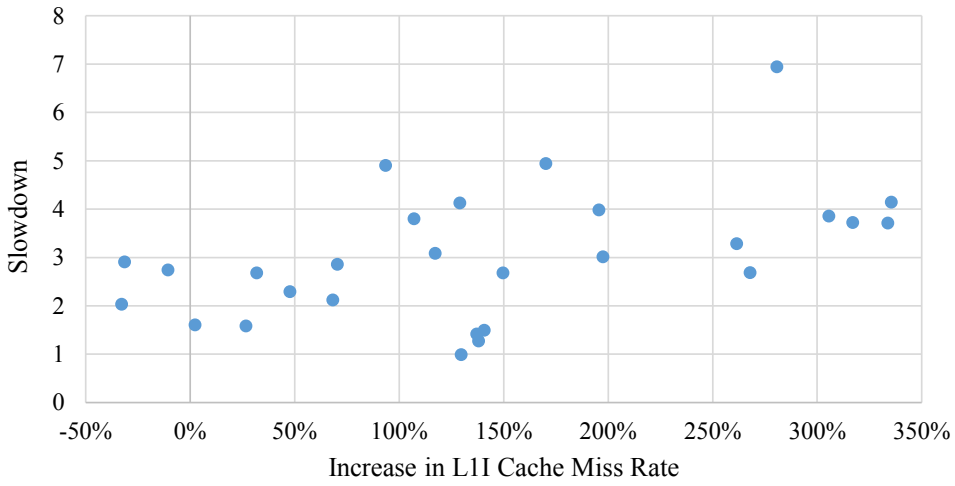


Figure 20: Correlation between Runtime Slowdown and the Change in the L1I Cache Miss Rate

are the inline instrumentation. One extreme example of this increase is that the instrumentation for each mutex is inlined, at the cost of 22 instructions which total 103 bytes versus 6 instructions for 23 bytes to load the mutex address and call the pthread library. During runtime, the dynamic instruction count is only increased by an average of 100% (and down from 120% with the TACO instrumentation) across PARSEC, SPLASH and NAS; and in this respect, *fluidanimate* is near the mean at 95% increase.

Figure 20 shows how the change in the L1I cache miss rate relates to the runtime slowdown from Contech. There is some correspondence between the two quantities, and the most indicative of the actual slowdown from the instruction execution related metrics. At less than double the instruction cache miss rate (100% increase), all benchmarks showed less than a 3x slowdown. And above the 150% increase in miss rate, the benchmarks almost all above the 3x slowdown. The CPI is primarily related to the data cache load and store miss rates, which are discussed in the next section.

4.6.2 Data Cache Miss Rates

A basic block event is the most common event (at least 18x more common than the next event), so I will use it to model the aggregate of recording all of the events in a program. To record the average basic block, which has a load and a store, there are 5 or 6 additional memory accesses: load the location of the (OS-managed) thread local structure, load the pointer to the thread local buffer, store the basic block ID, store memory access 1 and 2, and store the new buffer position. The optimizations in Section 4.2.1.3 eliminate 2 - 4 loads from each basic block event. With an average basic block event size of 14B and cache line of 64B, every 4th or 5th basic block event will incur a cache miss, which is 1 miss for every 16 stores or 6%. Note that the changes to the basic block event type have reduced the event size and the redundant address identification eliminated storing some memory operations.

Given that the Contech instrumentation incurs approximately a 6% L1 data cache (L1D) store miss rate, workloads with lower miss rates will have an increased miss rate from the instrumentation. Figure 21 shows how workloads with low native miss rates increase toward the 5 - 6% for Contech instrumentation, while workloads above this miss rate decrease. For example, a workload, such as *ocean_ncp*, with a large working set (in this case, 7GB) and high L1D miss rate (50%) will have an improved miss rate from the instrumentation. Effectively, an instrumented program has a second program (the instrumentation) interleaved throughout and this instrumentation has

While the *perf* results report a high L1D load miss rate for Contech, I devised a separate experiment to test the counters, where a simple program iterated millions of times by storing to a large buffer, but never loading from the buffer in the loop. *Perf* still reported L1D load misses, at a few percent above the number of L1D store misses. It appears that the underlying implementation of a "load miss" in *perf* includes the store misses as well.

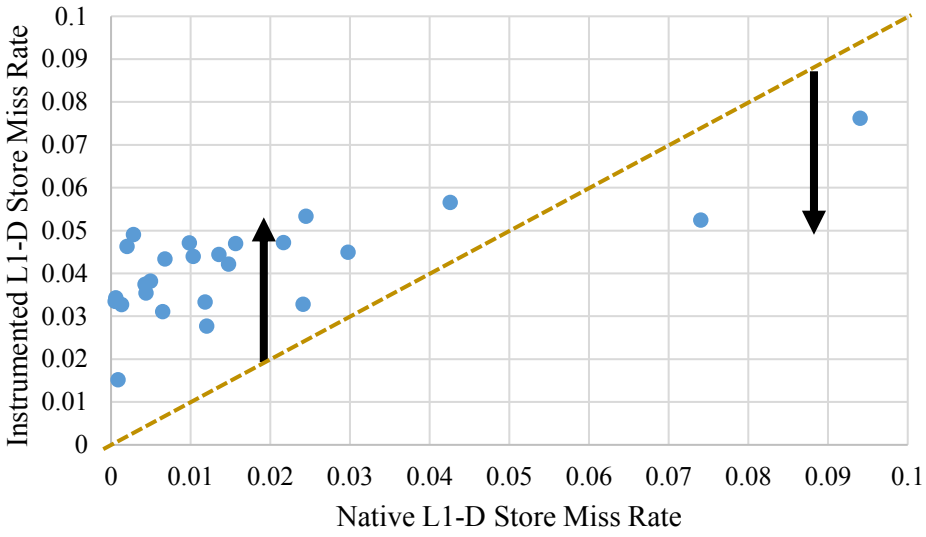


Figure 21: Relationship between Native and Instrumented L1 D Cache Store Miss Rates

a separate usage profile for the cache and other resources. And the amount that this "program" affects the base program correlates with the runtime slowdown.

4.7 Variation on Workload Parameters

For PARSEC, SPLASH and NAS workloads, the intent is always to maximize the number of workloads running successfully while utilizing as much of the system resources as possible. There are two parameters to tune this: size of the input and number of threads. Increasing the input moves the workloads closer to realistic execution, while also increasing the time and resources needed to run each parallel program. Using more threads decomposes the problem into smaller parts while increasing the parallel aspects of the program's execution: locks, contention, et cetera. Finally, each benchmark suite is uniformly run with the same set of parameters, such that even if, for example, *blackscholes* can run with native input, it will be run and compared using simmedium, just as the other PARSEC benchmarks are executed with this input size in order to generate maximal coverage of the suite.

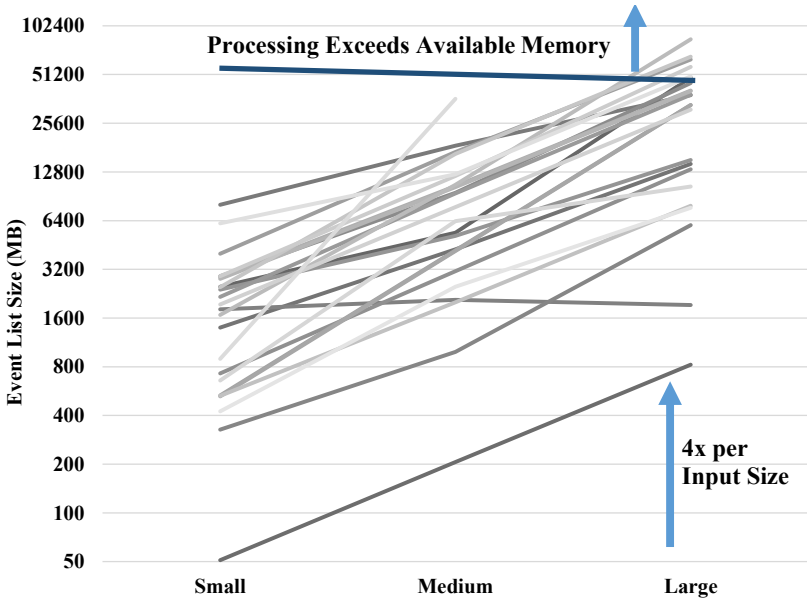


Figure 22: Size of Event List Generated from PARSEC and SPLASH with Different Inputs

4.7.1 Increasing Input Size

Every program has an initialization cost that is not completely accounted for with the runtime comparison. For most programs, this cost is low enough that it is subsumed in the existing sources of measurement error. However, at the smallest inputs, the program execution is so short that this value is noticeable, along with the other measurement errors. As such, no slowdowns will be shown below simsmall input size.

Changing the input size gives three insights into Contech. First, workloads that fail to generate a task graph (such as, *raytrace* or *water_nsquared* with the simlarge input), are only failing due to the size of the input and not any limitation in the design of Contech. Second, the failure to generate a task graph, or at least do so without reaching the instrumentation memory limit, is linked to the size of the event list. Given that the event lists size is increasing by approximately 4x per input set, this increase pushes the event lists for 7 workloads at simlarge beyond the available memory in the system, leaving only 17 workloads successfully generating task graphs. These "failing" workloads used more than 48GB of memory while generating the

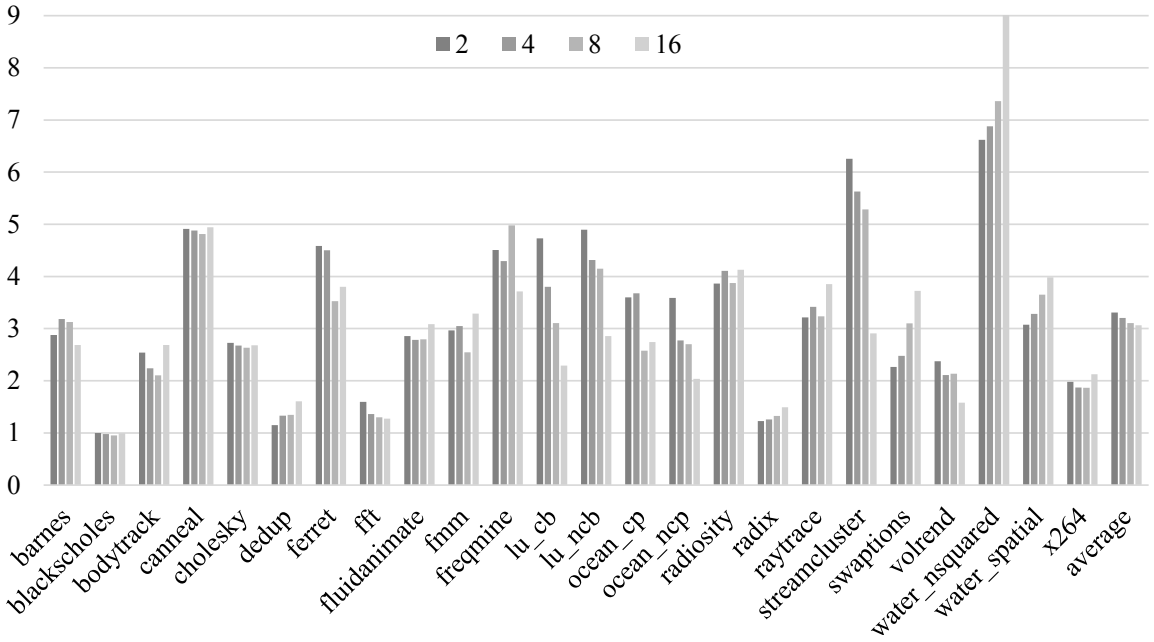


Figure 23: Runtime Slowdown of Contech Instrumented PARSEC and Splash 2 workloads by Varying the Number of Threads

event list, forcing their execution to be paused to allow buffers to be flushed to disk. Three of these workloads completely failed to generate task graphs when the local hard drive ran out of available space. Thus for the systems available, simmedium is the largest input for which all workloads run successfully to task graph generation, and hence why it is used for all other measurements.

The third insight is that the overhead of Contech depends on the program's code and not its input. The runtime slowdown showed small increase between simsmall, simmedium, and simlarge. The slowdown is primarily related to the the instrumentation's impact on the program; however, the longer trace has a small correlation with additional slowdown. Therefore, if a system had hundreds of gigabytes to terabytes of memory, then it would be able to record much longer executing programs.

4.7.2 Thread Counts

Figure 23 shows how the runtime slowdown varies as the number of threads specified changes. The overall average across the workloads trends downward, indicating that

Contech's design is scalable at least to 16 threads. Some workloads, such as *swaptions*, show worse scalability. *Swaptions* is task parallel, having no synchronization besides the thread creates and joins. The only constraint on its execution appears to be the data generation rate, which increases modestly between the 8 and 16 threaded execution. Effectively, the OS cannot allocate thread local buffers fast enough for Contech to fill them. As shown in Figure 14, if the buffers are preallocated, then *swaptions* executes faster especially at higher numbers of threads and the scalability loss from increasing thread counts is significantly decreased.

4.8 OpenMP Support

OpenMP support is a valuable addition to the Contech framework. It extends the task graph representation beyond the initial pthread-based model. This support also allowed Contech to instrument other benchmark suites, including NAS [66] and Rodinia [31]. This support was not without issues, which this section will address. The first major issue in supporting OpenMP is that the OpenMP pragmas are not stored in the LLVM IR, but rather have been translated into the appropriate code. The second major issue is that OpenMP's implementation relies on having 1 thread per processor (or the `OMP_NUM_THREADS`) allocated and then distributes work to these threads.

In most cases, there is a 1 to 1 transformation from the pragmas to the actual IR; however, OpenMP only defines the pragmas and their behavior while leaving the underlying implementation to the compiler / runtime. Therefore, more complex pragmas must instead be compiled and then the resulting IR examined to identify the routines and transformations used by the compiler and runtime. Furthermore, as each compiler is different, Fortran code using dragonegg and gcc invokes different routines than the code generated using OpenMP Clang compiler [61] (which targets

the Intel OpenMP runtime[3]), such as `GOMP_atomic_start` versus `__kmpc_single`. After the implementation routines are identified, it is generally a matter of adding them to the table of function names to instrument.

The second issue is more difficult. As the OpenMP runtime executes the program, parallel regions are assigned groups of threads by the runtime. Each thread in the group is treated as identical, although each can be distinguished within a group. Furthermore, in assigning a group, it is not clear how many threads will be in the group, until the group has been created and the threads have started executing. However, within these constraints, the runtime's threads are still OS recognized threads and therefore the thread local storage is preserved, including the context ID and thread local buffer pointer.

A parallel region begins executing with the invocation of `__kmpc_fork_call`, which takes in a function that is called by each thread assigned to the group. Since the number of child threads is only known after the parallel region begins, each child thread is passed additional information so that it can record its create event on behalf of its parent context. And given that context IDs are only stored with the buffer meta-event, each create event also requires queuing the current buffer so that the buffer event is also created. Passing and recording this information requires writing a wrapper for the function called in the parallel region. Furthermore, as threads are constantly being repurposed, the assigned context ID must be preserved and restored at each point where OpenMP may switch its execution.

The design for a parallel region exposes architectural information via the assigned number of threads. This is true even in with `omp parallel for` where the parallelism is over the iterations, as the compiler front end transforms this pragma into two sequences: `omp parallel` and then `omp for`. As Contech independently instruments each basic block, the split pragmas are instrumented separately, which results in

[3]Intel's OpenMP runtime also includes wrappers for the gcc-based routines.

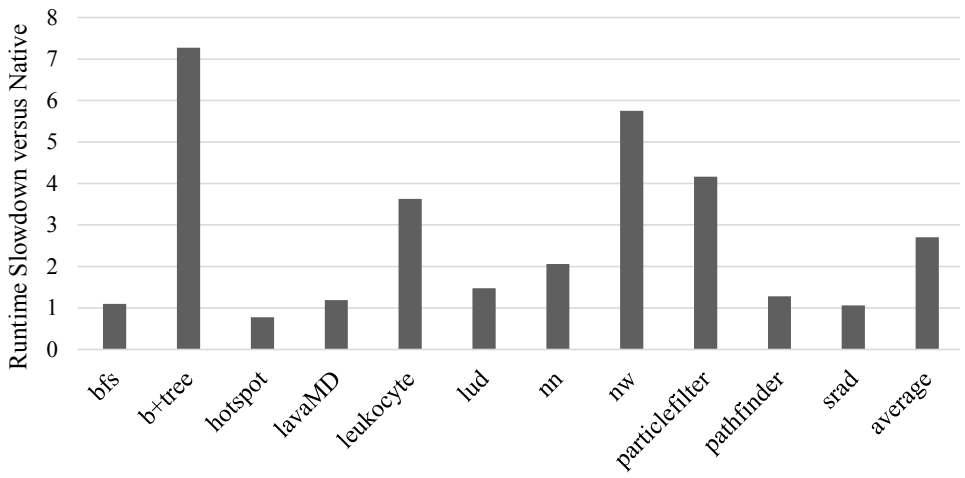


Figure 24: Contech Instrumentation Performance on the Rodinia Benchmark Suite

viewing the for loop's parallelism in two stages: creating the threads for the parallel region and the parallel iterations executed by each thread. While the stages could be combined when creating the task graph, this would special case on the assumption that programmers generally write parallel for rather than for loops in larger parallel regions. Furthermore, showing the iterations by thread can reveal thread imbalance, which is a common issue in OpenMP parallel loops.

4.8.1 Rodinia Benchmarks

The Rodinia benchmark suite [30] is designed to provide both GPU and CPU versions of computation kernels, using CUDA and OpenMP respectively. This permits the exploration of the heterogeneous computing space. At this time, Contech has not been extended to support GPUs; however, the CPU versions provide an additional set of benchmarks. Figure 24 presents the runtime slowdown from a subset of the Rodinia benchmarks. As the suite was further designed to be diverse, the slowdowns are also across a wide range, although the average is 2.7x.

4.8.2 LULESH

LULESH [72] is a leading proxy app from an effort out of the Lawerence Livermore National Lab, which models C / C++ scientific applications. Due to the high context creation (over 1.8 million), Contech's overhead applied to creation events significantly slows the program's execution to almost a 10x slowdown. Each task (such as a parallel loop iteration) only executes for an average of 15 basic blocks, which instead adds significant overhead from the context creation instrumentation forcibly queuing buffers. In queuing these minimal sized buffers, the small buffer copy path avoids wasting memory; however, the global lock on the common buffer list has measurable contention.

4.9 MPI Support

While the focus of the instrumentation design has been on shared memory programs, Contech also supports distributed memory programs that use MPI to communicate. Each instance of a distributed program generates a separate trace. The middle layer can then merge the traces together into a single task graph. The task graph has an empty task 0:0 added as the root that will create the instances of the parallel program. In order to maintain the address space separation, each memory address is prepended with the instance of the program (i.e., the rank), thus treating the program as if it used a partitioned global address space (PGAS). As sending and receiving data is an explicit operation, the ordering of sends and receives is represented as *sync* tasks, along with additional memory operations to denote the transfer of data between the two address spaces.

For each `MPI_send` and `MPI_recv`, a 3-tuple is created: the source rank, destination rank, and tag. This information along with the size of the operation are recorded for each request. These tuples are used to match the send and recv requests between ranks and thereby combine the different task graph pieces into the single

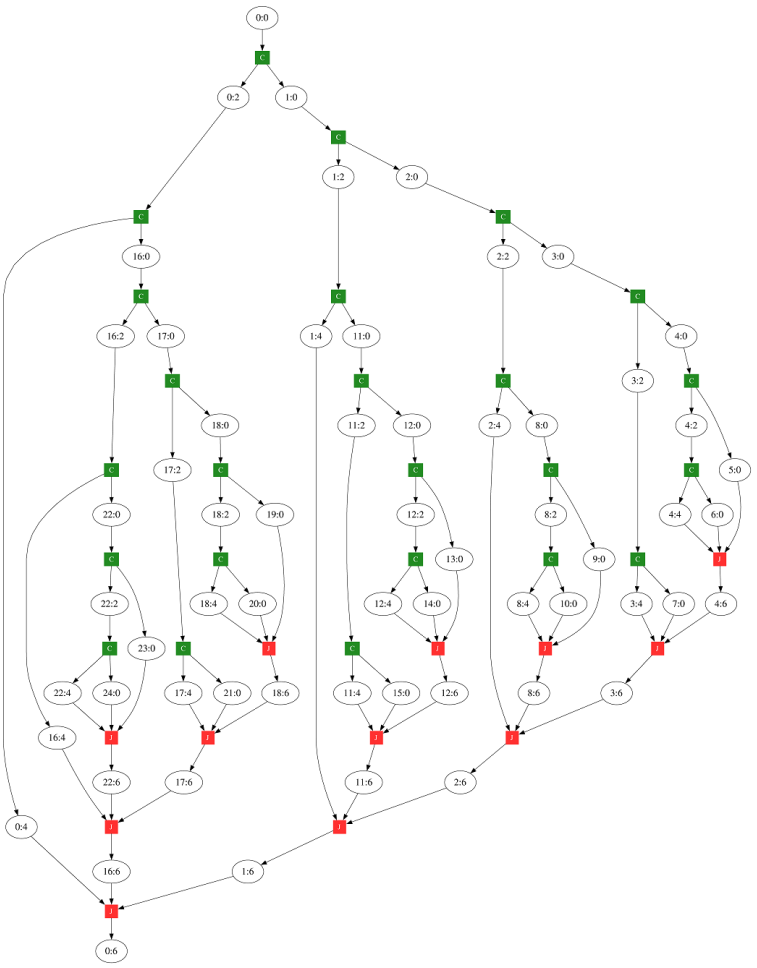


Figure 25: Task Graph Recursively Computing the 6th Fibonacci Number using Cilk

comprehensive program. To support these routines, Contech must invoke MPI functions; however, it is undesirable to always link in the MPI library, so Contech has two versions of wrappers around the MPI functionality.

4.10 Cilk Support

Cilk support is also under development for Clang / LLVM [60]. Cilk is particularly designed to minimize the difference between the serial and parallel stack sizes. As serial execution will follow each function call in turn, making these calls parallel via `cilk_spawn` has the effect of Cilk following the calls and leaving the continuation of the calling function as parallel work. As a consequence of this design decision,

the Cilk design utilizes `setjmp` and `longjmp` as the underlying mechanism to handle the parallel invocations. Cilk's design is the opposite of Contech's, in that Contech expects a thread to create parallel work that other threads will execute. As such, Contech internally must swap context identifiers between the threads as they create and execute the spawned work.

Figure 25 shows an example task graph from an example Cilk program that is recursively computing numbers from the Fibonacci sequence. This simple program is sufficient to exercise the compiler-based identification of the Cilk calls, as well as the different cases involving work stealing. The initial Cilk compilation inlines significant parts of the functionality, which contrasts with the OpenMP approach that uses function calls, and consequently, an operation such as `cilk_spawn` is transformed into several basic blocks. The `cilk_sync` call has also been inlined and furthermore may be executed implicitly by Cilk determining that this frame has no further work to execute. Therefore, Contech relies on Cilk's determination that the parallel work is complete. The support was subsequently tested on a million-element parallel quicksort used to test Cilkview [55]; and while technically correct, there remains significant future work to bring the overhead of this instrumentation support in line with the other paradigms.

4.11 Other Implementation Issues

At the start of a program's execution, the OS and the language runtime work to load and initialize the program. One of the required steps is initializing static variables. Many statics are initialized to zero and directly receive zeroed pages. More complex are the variables that are instances of a class. The initialization of an object requires invoking its constructor, and as the constructor was compiled as part of the program, it is instrumented. Contech's design is to encapsulate the program's entry point, `main`

and initialize the instrumentation system just prior to that point. However, instrumented constructors are one instance of code that may be invoked earlier than this point. Initially, every instrumentation routine checked whether a thread local buffer was present before storing the event, but that was incredibly expensive. Instead, the Contech runtime initializes the thread local buffer to a special static buffer. This buffer acts as a sink for instrumentation events. The queue buffer method checks for this buffer and will discard its events rather than globally queue it.

Contech task graphs include a notion of absolute time via the timestamp counter. Each processor's timestamp counter ticks independently of the others. Even though modern processors are often designed to maintain a constant tick rate, small variations in manufacturing, clock speeds, temperatures, et cetera can result in differences (i.e., skew) between each counter. Given the usefulness of the timestamp counter, a modern OS may regularly synchronize the counters to minimize the variations. Still, given the fine-grained nature of the timestamp, any measurement of the skew can only achieve a minimal bound. Contech has support to estimate the skew between threads; however, measurements showed that the OS was minimizing the skews and therefore the cost of this measurement provided minimal benefit.

4.12 Contech Compiler Wrapper

Each source file is passed to a compiler wrapper script. Contech must inject the additional step of invoking its pass on each source file during compilation; however, as far as is possible, the script leaves the compilation step unchanged. The script also wraps several different languages and selects the appropriate compiler steps to generate LLVM IR for instrumentation. Modern compilers may also be invoked as linkers, in which case they pass the link commands along. The wrapper script must do the same; it detects when the user intends to link the files into a binary which then requires appending the Contech runtime and static basic block information to

the linked objects. The static basic block information is then part of the initialized data in the binary, which the background thread writer accesses and writes directly to the event list.

As Contech uses `-flto` as part of compilation and linking, this flag introduces additional issues. With the `-flto` flag, Clang will emit LLVM bitcode (LLVM IR in binary form) rather than native assembly. These "object" files may then be merged into an archive, or other intermediate step before being linked into the final binary. In some cases, successfully compiling a benchmark has required changing the compilation steps to avoid one of these problematic cases. Consequently, Contech is not always a drop-in replacement for the current compiler, although it is more often an artifact of LLVM's limitations than the design of the compiler wrapper script.

Three compiler passes are required for each source file, as Clang does not support dynamic loading LLVM passes. The first pass uses Clang to generate the LLVM bitcode. The bitcode is passed separately through *opt*, which applies specific LLVM passes (in this case, Contech's instrumentation) to bitcode files. Then the instrumented bitcode is passed back through Clang to optimize some of the applied instrumentation, such as merging basic blocks that Contech split. These steps increase the time to compile each application by an average of 68%.

4.12.1 Undefined Behavior

As part of successfully compiling programs with Clang / Contech, I have encountered a variety of compiler and linker errors. The most common is subtle differences between `gcc` and clang. Many programs are written targeting `gcc` as their compiler, which has given particular meaning to certain undefined behaviors in the C standard against which Clang takes different approaches. This can force a programmer to expend additional effort addressing these cases. For example, many programs in C do not specify a return value from `main` or their pthread worker routines. `Gcc` will return 0

Table 6: Decision Forest Regression Model Parameters

Parameter	Value
Resampling Method	Bagging
Create Trainer Mode	Single Parameter
Decision Trees	8
Maximum Depth	32
Random Splits	128
Minimum Samples	1

or garbage values in these cases. In contrast, Clang uses these cases to optimize and treats the end of the functions as unreachable, which will consequently result in the program failing to execute to completion. Quoting from the C specification: [63]

> Flowing off the end of a function is equivalent to a return with no value; this results in undefined behavior in a value-returning function.
>
> If the } that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined.

4.13 Predictor of Contech Performance

This chapter laid out the the design of Contech's LLVM-based instrumentation toward the twin goals of having diverse support for parallel programming languages and paradigms, as well as minimizing the impact on parallel programs by having low overhead.

The continual question is what program behavior corresponds to low versus high overhead with Contech instrumentation. Understanding this distinction provides insight into the specific costs of the instrumentation, as well as some observation into the gap between the parallel program's requirements and the capabilities of the architecture. Using Microsoft Azure Machine Learning tools [89], it was possible to derive an approximate model of the runtime slowdown based on measurements taken during execution. Initially starting with over 50 measurements and hardware performance counters from the native and instrumented executions of all benchmarks (set to 16

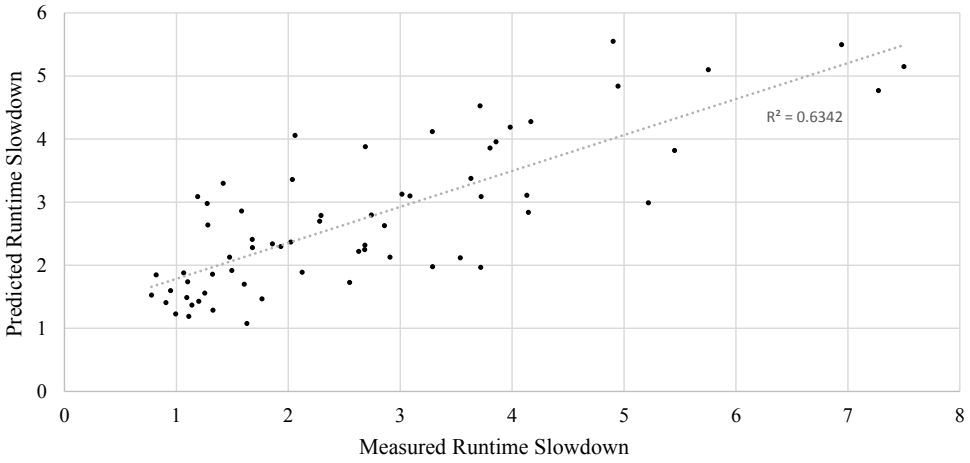


Figure 26: Approximate Fitting of Runtime Slowdown based on Other Metrics

threads and PARSEC workloads ran with simsmall and simmedium input sets), the data was refined to seven metrics (see below) that enable the best fit of the slowdown, as well as processed through different model types (such as, linear regression, neural network, and decision forest) where the decision forest [33] produced the best fit. The decision forest model was configured with its default parameters, as shown in Table 6.

Each model was run through 90/10 train and test splits of the data, with the results of the best model (decision forest) shown in Figure 26. From the model, we can observe both workloads having unusual characteristics, as well as those characteristics that best describe (and likely contribute) to the cost of the instrumentation. Contech's instrumentation of *NN* from Rodinia increases the dynamic instruction count around 6x, as the workload creates many small tasks that require Contech to record and track; however, its slowdown is only 2x. The instruction cache is significantly affected by the instrumentation for *Raytrace* with the simsmall input, as the miss rate increases 24x even as the slowdown is 1.67x and half of the simmedium slowdown. And in general, the model was the most accurate for PARSEC with simmedium.

As noted earlier in this chapter, the single best metric for modeling the runtime slowdown is the size of the event list produced from the instrumented execution. Similar to the event list, the increase in the dynamic instruction count can further refine the modeled slowdown. As the programs are parallel, the dynamic instruction count is also influenced by the quantity and behavior of parallel threads, unlike sequential execution where this value would be tightly coupled to the execution time. The estimation of the runtime slowdown can be further refined by incorporating the frequency of new tasks, commonly from synchronization. In Section 4.6.1, one of the metrics was the increase in the L1 instruction cache miss rate, which had related to the runtime slowdown. Two metrics are also taken from the native execution: the L1 data cache store miss rate (see Section 4.6.2) and the backend stall cycles per instruction. This latter measurement corresponds to the baseline program's resource usage and capability to absorb the instrumentation's additional requirements. Finally, the parallel programs have more slowdown from the instrumentation when their basic blocks have fewer memory operations in them. While several metrics come from the baseline program, those alone are not sufficient to estimate the impact that instrumentation will have on the program.

Therefore, while the slowdown for most parallel programs averages 3x, there are particular program characteristics and patterns that can drive the cost of instrumentation higher. In addition, the instrumentation is clearly stressing the capabilities of modern architectures, while lighter-weight designs such as ARM are more severely impacted. It remains for future investigations to determine whether this overhead is transparent enough for analysis purposes, as well as to further identify the microarchitectural limitations encountered by the instrumentation. In the following chapters, I will explore the process of analyzing Contech task graphs and then model a complex system requiring the full capabilities of the task graph.

CHAPTER 5

PRACTICAL TASK GRAPH ANALYSIS

This chapter will explore the analysis of the task graph representation as generated by the instrumentation in the prior chapter. The components of the prior chapter work together to produce a rich representation of the execution of a parallel program in the form of a Contech task graph. It is vital that the representation be usable and useful, and so I will cover the basics of how to access the task graph, common concerns and issues, as well as several examples. The following chapter will use the task graph to help understand a complex parallel program modeling problem.

Contech task graph analysis is done via programs termed, backends or back ends. The analysis of a Contech task graph is separated from the instrumentation and generation of the task graph. This separation is purposeful so that the instrumentation is not additionally burdened by analyzing the program's events while it is running, to reinforce the idea that task graphs are independent of the architecture generating them, and furthermore to better support diverse program analyses and the possible synergies between them.

5.1 Back end tools

Contech's purpose in collecting the rich task graph representation is to support a wide variety of back end tools that can characterize the behavior of a benchmark and/or collect metrics by analyzing a task graph. The task graph file contains three elements: the debugging information, the tasks of the graph, and the table of contents that provides a breadth-first traversal and random access to the tasks. Back end tools' traversal of the task graph is supported by Contech's task library API, which is written in C++11. The first step is to instantiate a `TaskGraph` from the file.

```
TaskGraph* tg = TaskGraph::initFromFile(fileName);
```

Tools will commonly iterate through each task in the graph, following the breadth-first traversal. A tool can also request a specific task using `getTaskById(TaskId)`. Or access the region of interest (ROI) task IDs by `getROIStart()` and `getROIEnd()`, and thereby restrict its analysis to the ROI.

```
while (Task* currentTask = tg->getNextTask())
```

The task library API enables programs to access the features of the task graph, and enables iteration via a set of classes over the components of interest from a task graph: tasks, basic blocks, and memory operations. Commonly accessed properties are the type of task, as well as the tasks start and end time.

```
switch(currentTask->getType())
```

Most of the data in a Contech task graph is part of the work tasks. Each work task contains a sequence of basic blocks executed, as well as the memory operations performed by the instrumented code.

```
auto bba = currentTask->getBasicBlockActions();
for (auto bbIt = bba.begin(), bbEt = bba.end(); bbIt != bbEt; ++bbIt)
    BasicBlockAction bb = *bbIt;
```

The API also exposes "debugging information" to back ends, such as function names, file names, and line numbers. This information is contained in `TaskGraphInfo` class.

```
TaskGraphInfo* tgi = tg->getTaskGraphInfo();
```

Most of the debug information is currently associated with basic blocks, in the `BasicBlockInfo` class.

```
auto bbi = tgi->getBasicBlockInfo((uint)bb.basic_block_id);
```

By using the task graph, a back end might simulate branch behavior, caches and coherence, analyze synchronization usage, or find data races. Coherence modeling and data races detection will be revisited in the next chapter, as part of modeling parallel programs. Cache simulation has been published previously in TACO [100] and data race detection with Contech was first published by Philip Vassenkov [121]. Other example analyses will be discussed in Section 5.3.

5.1.1 Types

Contech's task graph API, described briefly above, relies almost exclusively on three sets of types to convey the content of a task graph: IDs, actions, and tasks.

5.1.1.1 IDs

Section 2.4 described the visualization of a task graph, and one of the first aspects is the unique label applied to each task, its task ID. All tasks in a Contech task graph are uniquely identified by their task IDs, or `TaskId`. This type has two component types, `ContextId` and `SeqId`. Technically, the `TaskId` is a 64-bit value; however, the API is designed to use the type system to minimize the ability to access the underlying representation.

5.1.1.2 Actions

Every basic block execution and memory access is stored as an *action*. Given the potential to store billions of these *actions* in a single task graph, it is vital to select an efficient representation. Taking a 64-bit value, basic blocks only use the lower 23 bits and memory addresses the lower 48 bits, which leaves 16 bits available. The `Action` type is a bit-field that merges basic block IDs, memory accesses, and other specific events (such as `malloc`) into a single type. The first use of the spare bits is to include a type field, enabling the values to be distinguished, including whether the access is a read or a write. Shadow Replica [65] also encoded other execution

information using the spare bits and otherwise invalid addresses. Spare bits are also used to encode the size of a memory access, which are always powers of 2. Memory accesses may also need additional bits to specify the MPI rank of the address, which enables the distributed memory program to be treated as if it used shared memory (see Section 4.9).

By unifying the actions into a common type, Contech can utilize C++ standard template library (STL) data structures to store and manage the actions internally. Specifically, actions in every task are stored in a single action vector. The iterators shown previously operate on top of this vector and provide action type specific access, such that a backend does not need to check every type and instead can be simplified to operate on basic blocks or memory addresses, if that is all the backend requires. Using common sized types, this solution "wastes" 4 bytes per basic block; however, other approaches would likely require an extra pointer. Given the hundreds of millions of basic blocks in the average PARSEC task graph, any storage scheme must be cognizant of the need to minimize the space used.

5.1.1.3 Tasks

Tasks are composed of a task type, which is the associated graph partition: create, join, sync, barrier, and work (or basic blocks), as well as the sequence of basic block and memory actions, timestamps for the start and end of the task, and two sets of TaskIDs for the predecessors and successors in the graph. While tasks are the predominant type in the graph, the majority of analyses focus on the actions it contains. Even sync and barrier tasks contain an action, which is the address of its associated object. Commonly, the address serves as a unique identifier for managing information about the object, as the task graph has already placed the appropriate dependencies between syncs on the same address.

5.1.2 Determinism in Analysis

Each analysis is applied to a task graph, representing a single instance of the program's execution. Similar to other dynamic analyses, improving the quality of results in an analysis with Contech may require collecting and analyzing multiple task graphs to provide different traces of the program's execution. Most analyses are deterministic and so repeated invocations will not generate different results. For example, the same task graph will always give the same sequence of memory accesses, when following the same partial-ordering of all tasks.

Some tools, such as MACPO [101], incur additional runtime overhead in the effort to (attempt to) order all memory accesses from a parallel program, although the effect of the instrumentation and the races to add accesses to the single log may prevent the trace from representing the true, executed order. Furthermore, any instruction stream may be further impacted by out of order execution, consistency models, and other microarchitectural effects. Similarly in analyzing a task graph, it may be desirable to observe a single stream of memory accesses from the program, rather than the default of using them on a task by task basis. Naïvely, an analysis would round robin across active tasks, selecting a memory access from each. However, this approach would not account for differences in the instruction streams as well as the cost of individual memory accesses. In the other extreme, the analysis could simulate the entire program and thereby compute an order of all events in the program.

One alternative was utilized to reproduce the results from Harmony [70]. Each work task has a total duration and contains a quantity of basic blocks. For simplicity, this model assumed equal cost to each basic block, which then enables the blocks to be placed into a total order. From this order, the memory accesses in each basic block can also be ordered. Using the approach outlined in Section 4.2.1.1, the specific time required for each basic block can be estimated. Figure 27 covers the average and median cycle times for basic blocks, where the function calls and other high cost

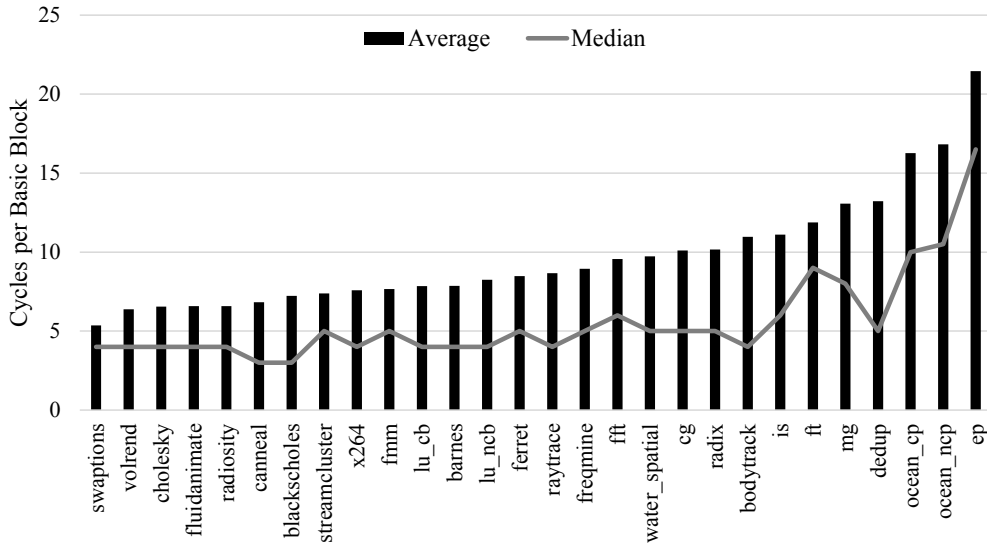


Figure 27: Average and Median Basic Block Cycle Times

blocks have been excluded. The average varies from 5 to over 20 cycles per block, while the median for most workloads is between 4 and 5. Thus assuming equal cost for each block is obviously simplistic; however, for many workloads the divergence will be reasonable, especially if there are frequent non-work tasks to resynchronize each context.

5.1.3 Basic Blocks

Parallel programs tend to execute the larger, measured in terms of memory operations, basic blocks more frequently than the smaller basic blocks. Figure 28 compares statically measuring the size of each basic block in the program with the average memory operations observed by running the program. *LavaMD* from the Rodinia suite has the highest dynamic average at 13.3 memory operations per basic block. *Water_nsquared* is an example of the opposite result, whereby its dynamic average of 1.07 is less than half the static average, perhaps as this benchmark has the largest static basic block of 133 memory operations, which skews the static average higher.

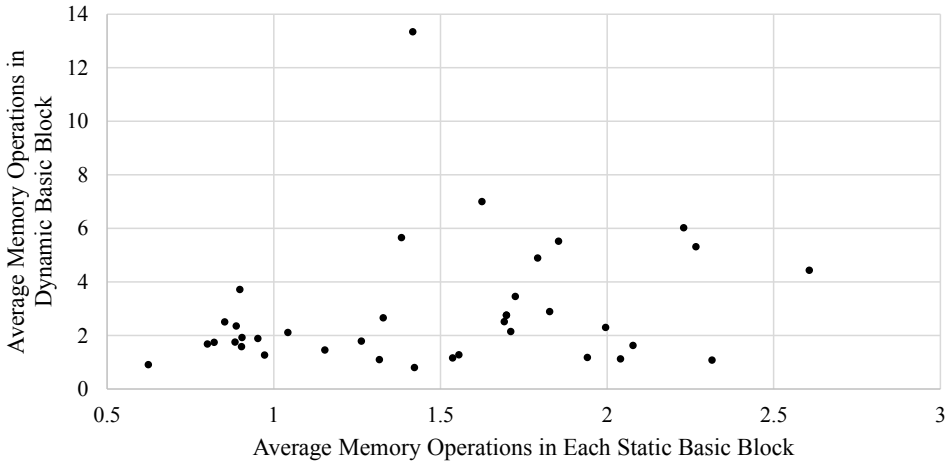


Figure 28: Average Static versus Dynamic Memory Operations per Basic Block

5.2 Parallel Programming Paradigms

Parallel programs are commonly written using certain patterns and paradigms. Patterns have hardware and software characteristics that can lead both to their identification [32] [37] [99], as well as exploitation [106]. Paradigms, such as pthreads or OpenMP, lead to particular ways of expressing the parallelism in programs. Pthread based programs add 1 context per create event, while OpenMP programs add between 3 and 60 per create (with a median of 16, which is the specified number of worker threads).

The OpenMP benchmarks from Rodinia and NAS rarely utilize fine-grained synchronization, instead they primarily rely on creating new parallel regions. *Cg* from NAS has 17 sync tasks per context on average, which is the highest of these programs, and has 58 parallel regions. In contrast, the median benchmark from SPLASH has 4400 sync tasks per context, as these programs utilize condition variables as barriers. Ignoring SPLASH's alternative barrier, the majority of the benchmarks use barriers, although the median is just 1 per create task, and for OpenMP programs this is often the barrier at the end of each parallel region.

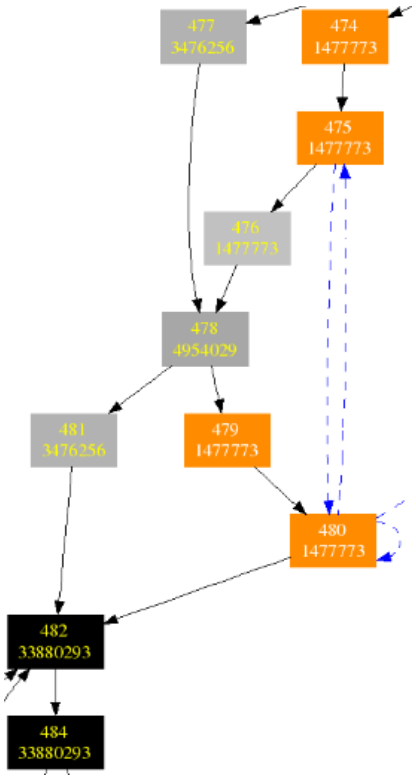


Figure 29: Subset of Dynamic CFG from Fluidanimate (BBID and Execution Count)

Within each suite, there are distinctions, such as the task parallel workloads *blackscholes* and *swaptions* that do not utilize locks nor barriers, or pipelined workloads *dedup*, *ferret* and *x264*. *Streamcluster* utilizes thousands of barriers, more than any tested workload except *lulesh* which has tens of thousands of parallel regions each utilizing two barriers. And at the finest-grained decomposition, *fluidanimate* and *raytrace* have millions of synchronization events during their execution.

5.3 Sample Analyses

The following section will explore three simple analyses of Contech task graphs to demonstrate some of the diversity of support available in using this representation of parallel programs.

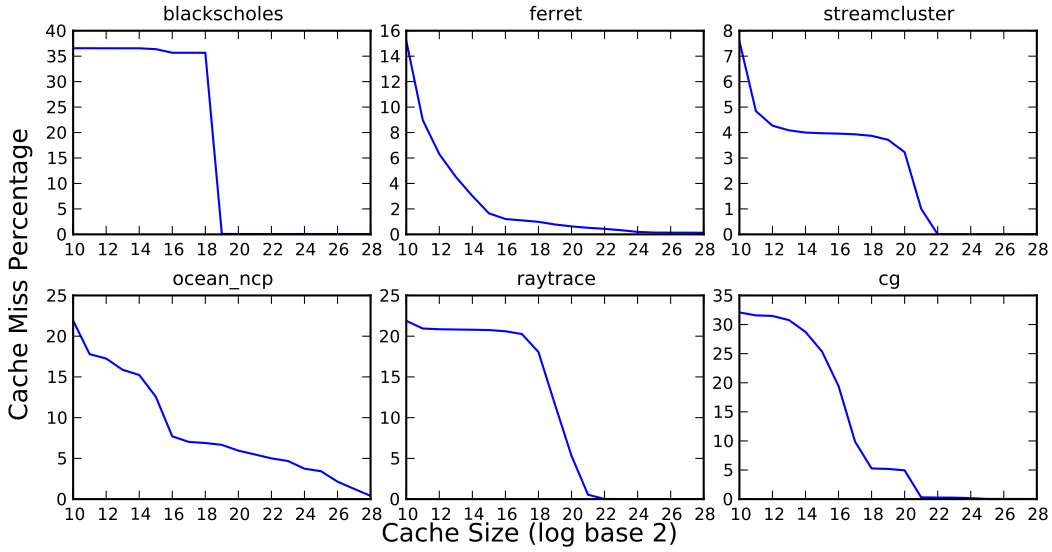


Figure 30: Miss Rate for Selected PARSEC, SPLASH-2, and NAS benchmarks with 16 threads, across shared caches sized 1KB to 256MB (2^{10} to 2^{28})

5.3.1 Control Flow Graphs

Figure 29 shows how the control flow from a Contech task graph can be directly visualized, by following the control flow of every context in the task graph. In the dynamic control flow graph, each node is a basic block (with its Contech-issued basic block ID) and has a dynamic execution count (and darker nodes are executed more frequently). Orange basic blocks are involved in synchronization, where blue arrows indicate data transfers / communication. Thereby showing one approach to understanding the main components of control flow and memory accesses together.

5.3.2 Cache Simulation

Contech also provides sufficient information to model architectural components such as caches and branch predictors. In Figure 30, we replay the memory access streams from previously collected task graphs multiple times as we iterate through the range of cache sizes, from 1KB to 256MB. All caches are 4-way associative and use LRU replacement. This analysis is deterministic and by using the same task graph and breadth-first traversal as stored in the task graph, the analysis will operate on identical

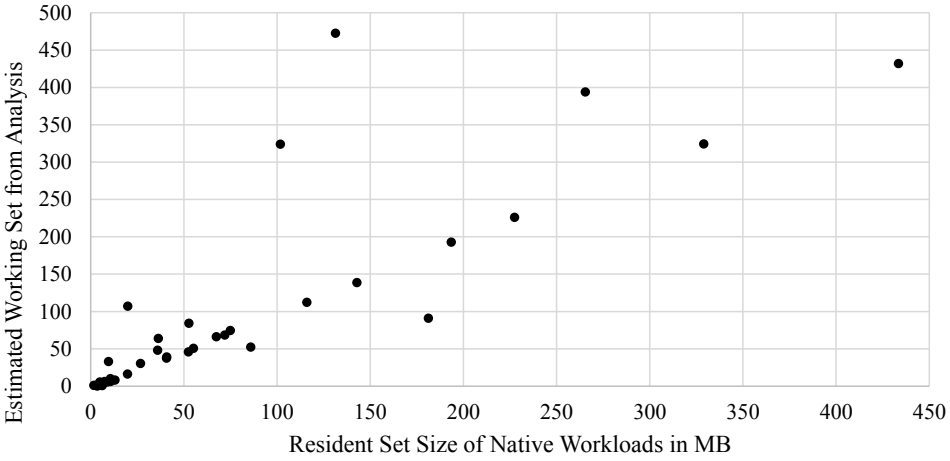


Figure 31: Estimated Working Set versus Actual Resident Set Sizes for Uninstrumented Benchmarks in MB

memory access streams, thus supporting the comparison of the different cache sizes. As with all dynamic analyses, the cache model could require analyzing multiple task graphs from the target program, possibly across diverse inputs, in order to have the necessary confidence in its conclusions.

The cache analysis models show the different characteristics of the benchmark's memory accesses and how increasing the cache size impacts the miss rate. Benchmarks such as *blackscholes* and *raytrace* show minimal improvement from increased cache size until a component of the working set entirely fits in the cache, with significant reduction in miss rates, which contrasts with *ferret* and *ocean_ncp* where the miss rate is continually decreasing from larger caches. *Streamcluster* and *cg* are workloads with multiple plateaus from different components of the working set dominating the cache miss rate.

5.3.3 Memory Usage

Similar to modeling a cache, Contech can also compute a program's working set by tracking the memory addresses accessed. Furthermore, as Contech captures memory management, it is able to split the working set into heap and non-heap accesses, where

the latter are both globals as well as stack addresses. Figure 31 presents the results of this analysis compared with the measured resident set sizes for the uninstrumented workloads. Resident set size gives maximum memory required by the program at any one time during its execution, due to paging and other effects this may be less than the working set of the program (and the one estimated by the analysis). In general, this shows that should the analysis be correct, Contech instrumented programs have similar working sets as they do without instrumentation. Two notes about these results: first, *ocean_ncp* has a resident set size of 7GB and was excluded so the plot could show the other 40 workloads. Second, in studying the working sets of instrumented programs, `/usr/bin/time` over reports the size by a factor of 4 and the results here have been scaled accordingly, and this is a known bug in `time`.

5.4 Summary

This chapter has covered both how Contech provides analyses with access to the task graph representation, as well as demonstrated several backends to analyze task graphs in different ways. These analyses show a sample of the diversity of analyses supported by Contech's framework. The following chapter will explore a comprehensive analysis that requires the complete representation.

CHAPTER 6

PARALLEL PROGRAM MODELING

In this chapter I will explore how Contech can be used to drive a comprehensive model of parallel programs. This usage will rely on every component of the task graph representation. This model will be used to explore a possible reconfigurable system, see Figure 32. This system seeks to address the architectural challenges first outlined in Section 1.2.1. In this system, the architecture provides the capability to reconfigure based on the explicit directive of the program. This enables hardware to dynamically adapt to the program's resource requirements, omitting the lag associated with a dedicated detector that discerns these requirements. The compiler will analyze the program and its source to determine the appropriate configuration for each section of the program. Particularly by using Contech's task graph representation, I will analyze parallel programs at the task granularity and show that even at this coarse grained unit of execution, programs differentiate and specialize their execution to specific resources. The initial results from this chapter have been previously published in collaboration with Seshan [110].

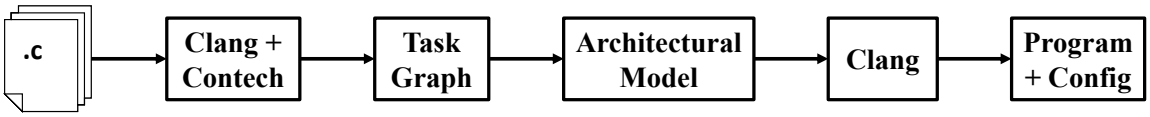


Figure 32: Proposed Reconfigurable System Workflow

6.1 Related Work

Parallel program modeling and even investigating heterogeneous or reconfigurable hardware has been extensively studied. This section summarizes the recent work that leads to the proposed model and system demonstrated in this chapter.

6.1.1 Modeling Parallel Programs

Adve and Vernon [2] showed that hand-constructed task graphs with basic per-task resource usage from hardware performance counters can predict parallel program performance. Their approach was limited in its task graph construction (see Section 2.1); however, adding information about each task's CPU usage and memory usage (cache miss rates, coherence events, etc) enabled them to accurately model the performance of parallel programs on real hardware. Besides the task graphs being hand constructed, the additional information about resource usage was collected from the hardware itself, rather than derived from a separate model.

Parallel program resource usage can also be modeled analytically using several different methods. LogP, first discussed in Section 2.7.3, provides a method to estimate the effect of hardware on the program based on the cost of data communication. The Roofline model [128] proposed viewing programs based on their computational and memory intensity. The initial model was based on a static analysis of the programs; however, it has been extended to utilize measurements of real executions [96]. Hardware can then be designed to balance the two costs [36].

6.1.2 Simulating Parallel Programs

Parallel simulation work [12] [26] [28] has emphasized splitting the simulation along temporal lines and reconstruct missing information, where as the approach in this chapter is to split the simulation spatially along the threads themselves and use targeted models to provide the missing details. Lauterbach [81] modeled single-threaded programs first with a detailed cache model and then used that model as an input

to simulation of instruction samples, showing that a program's simulation need not be done in a single pass. As a precursor to the task graph model, Rico, et al. [102] called the *non-work*: paraops, and showed that abstracting these operations supports a simplified simulation infrastructure [103] and can lead to speedups in simulation time.

Zhang, et al. [134] showed the importance of modeling communication behavior to accurately capture true parallel program behavior, in that measuring the on-chip traffic gave better indication of different phases of the program's execution which lead to identifying more representative samples.

Using a variety of processors as data points, Bhadauria, et al. [19] performed an analysis of the scaling and bottlenecks of PARSEC workloads as observed on real hardware. Lee, et al. [82] extract representative execution paths from a program's trace and use these paths for rapid design space exploration in order to find the optimal resource configuration for the entire program; however, their work does not address parallel programs, nor the reconfiguration of the programs while running.

6.1.3 Software-based Reconfiguration

In a reconfigurable system, the compiler / runtime is capable of selecting the appropriate issue-width [64], issue queue size [67], as well as establishing an IPC target for hardware [125]. Dynamic compilers have also been demonstrated selecting hardware resources [58]. Wu, et al. [130] introduced a dynamic component to programs that has better knowledge of the code to be aware of when the program is most likely to need a change to the configuration, demonstrating that informed software can outperform hardware-based reconfiguration even when accounting for the additional overhead from the software's analysis. Prior work has shown that the compiler can provide hardware with information about loop dependencies [111] and the loop body [48] to speedup execution.

Rather than allowing extensive reconfiguration, if the hardware platform is designed with pre-existing differentiation in configuration (heterogeneity), then the software can instead schedule the execution threads to the most appropriate hardware configuration. The particular benefit is that there is no need to change the ISA to support the reconfiguration information and therefore legacy applications can still benefit. ARM big.LITTLE [50] is one such hardware platform. With a diverse heterogeneous hardware, the software itself can be tailored by scheduling threads to specialized cores: based on critical sections [114], whereby the thread holding a lock is switched to the faster core or ISA usage [51], wherein many cores only support a subset of the ISA thereby saving decode power. With QsCores [122], the software selects the quasi-specialized units to execute on.

Saez, et al. [105] proposed a thread scheduling for asymmetric multicore systems whereby program threads are assigned speedup and utility factors based on collecting targeted metrics. The utility factor is the estimated impact of the potential thread speedup on the program's execution time, where the utility factor is equal to the speedup factor for serial programs. Using these metrics, the OS can then make new scheduling decisions each time a parallel phase completes, i.e., a barrier. Similar to this work, I assumed that sufficient cores were available for the program, which given the reconfigurablity of the cores obviates the need to model scheduling decisions. And rather than use online metrics, I explored how the compiler (through an offline model) could determine the appropriate resources for different sections of the program.

Jones, et al. [68] explores how the compiler can inform hardware about the lifetimes of registers to thereby enable the processor to reclaim physical registers earlier and thereby achieve equivalent performance with a smaller register file, which supports the claims herein that the compiler can provide specific insights into the program earlier than the hardware can detect them.

6.1.4 Hardware-based Reconfiguration

Hardware can monitor the program to detect phases and then adapt the resources accordingly; for example, buffer sizes [40] and integer versus floating point [104]. Rather than using phases, hardware approaches can also adapt after fixed-sized epochs: distributing SMT resources [126], or scheduling to specialized cores [93]. This work is focused on software-based schemes; however, these publications demonstrate the additional possibilities for reconfiguration and adaptation of the architecture.

Thread scheduling policies are important for maximizing performance even without diverse hardware resources. Wang, et al. [127] focused on the interaction between thread mapping schemes when scheduling multi-programmed workloads, and analyzes how the different scheduling decisions impact the resource usage particularly the shared caches and the prefetchers.

6.2 Reconfigurable Parallel Architecture

For this exploration, I propose an architecture based on recent Intel designs [43], which has additional execution resources, such as integer ALUs or floating point multipliers. By default the additional execution resources are power gated off. This architecture also exposes new instructions (or other mechanisms) that enable software to request that certain resources be powered on or off. When the resource is requested, conservatively I assume that the processor has minimal ability to execute instructions while it reconfigures. Either the core must flush its pipeline and wait for the reconfiguration to complete, or can possibly be instructed to execute a specific instruction, such as test-and-set until successful. Either way, the processor is not doing useful work during reconfiguration. Furthermore, these reconfigurable cores are in a multiprocessor fabric (e.g., a multicore processor) that supports each core executing under a different configuration, which effectively results in the system's execution becoming heterogeneous as the cores reconfigure independently.

Table 7: Baseline Function Unit Configuration

	Latency (cycle)	Number of Function Units
Integer ALU	1	3
Integer Mul	4	1
Integer Div	73	1
Floating Point Add	4	1
Floating Point Mul	4	1
Floating Point Div	26	1
L1 Loads	4	1 Port
L1 Stores	4	1 Port
L2	12	1 Port
L3	30	1 Port
Memory	100	1 Port

Table 8: Baseline Resource Configuration

Resource Type	Size
Reservation Station	56
Re-Order Buffer	168
Load Buffer	64
Store Buffer	36
Line Fill Buffer	10
L1 size	32KB
L2 size	256KB
L3 size	20MB
Cache Line	64B
Memory Word	8B
Fetch Bandwidth	4 ops

The baseline function unit configuration is shown in Table 7 and the other microarchitectural resources are configured based on Table 8, which together models a SandyBridge-like architecture. A study of that architecture [44] measured floating point divides as having faster execution than their integer counter parts. Small inaccuracies are immaterial given that the model is computing relative speedup.

On top of this baseline configuration, the processor has an additional set of dark function units and execution resources. These resources are what can be enabled and disabled as needed. Some resources, such as an integer multiplier, would be relatively simple to add, where as others such as additional memory ports would entail

significantly more complexity. For the purposes of this work, I am not considering this or other costs, although future work is exploring how to incorporate the various costs of the resources into the configuration decision as well as the value of further resource increases, such as two additional floating point adders instead of one.

As the individual core is assumed to have little to no ability to execute instructions while reconfiguring, it is important that reconfiguration is not undertaken lightly or arbitrarily. The task graph provides clear points to potentially reconfigure. Each work task, by definition, must be bordered by non-work. The non-work tasks, particularly the *syncs* and *barriers*, represent hundreds to thousands of cycles in which the execution is effectively waiting. This non-work wait time can then be overlapped with the time required to reconfigure the processing core, thereby waiting on both the existing non-work as well as the reconfiguration.

6.2.1 Reconfiguration Time

Figure 33 gives an example execution, where two threads of program contain 4 work tasks (ovals) and 2 non-work tasks (rectangles). There are additional rectangular boxes with the metadata in the binary that contains the ideal configuration for the core that executes the particular work task. For illustration, assume that the first tasks of each context begin at the same time (T0), take the same time to execute (T1 - T0) and that both cores that execute these two contexts take identical times to perform either reconfiguration (T4 - T1). In such a simplified scenario, context 1 has greater *wait* time due to executing its *sync* second, allowing complete *overlap* of reconfiguration - which for context 1 is to power down 1 floating point adder and power up one floating point multiplier - with task synchronization, ensuring zero *overhead*. For context 0, however, even post *overlap*, there is some *overhead* that the reconfiguration introduces, thereby inevitably delaying the L1 heavy work task.

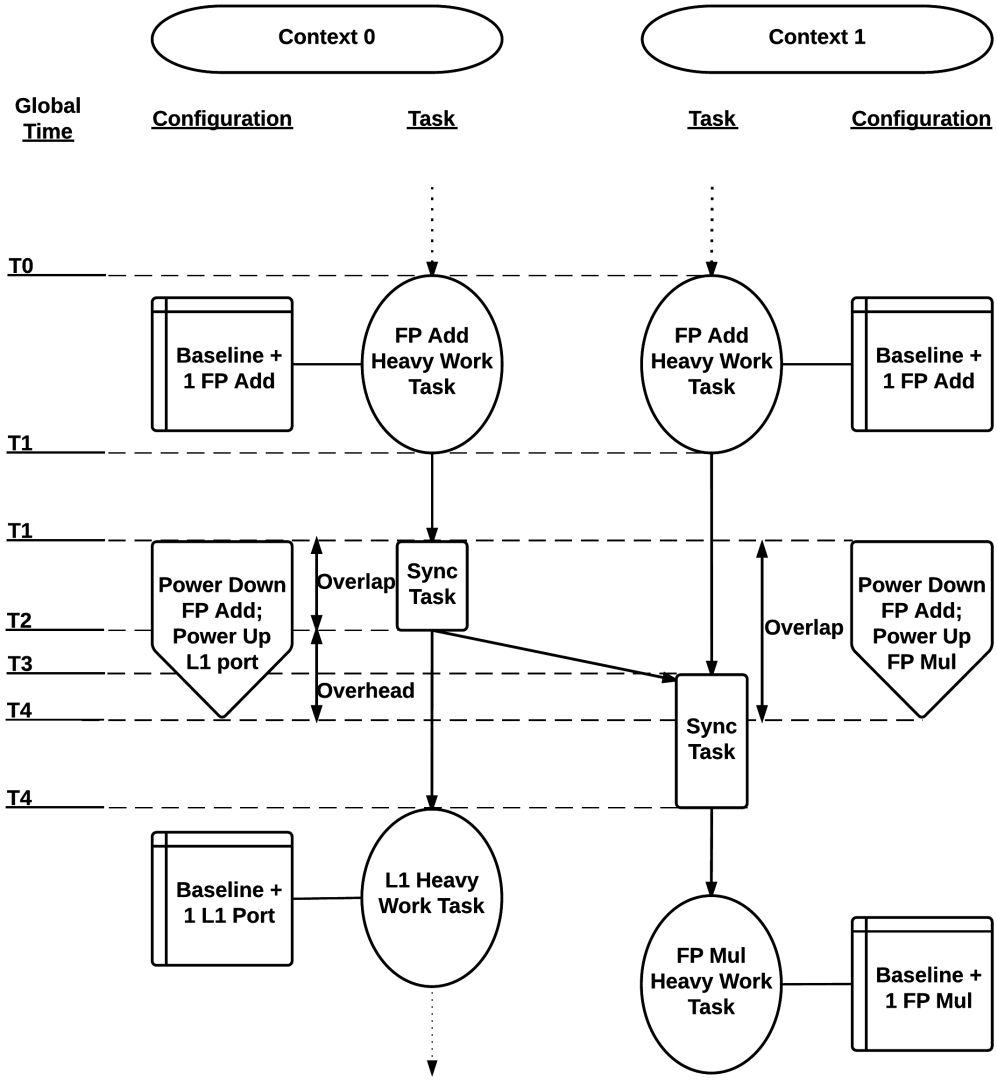


Figure 33: Overlapping Reconfiguration and Synchronization

In Section 6.5.5, I will explore the impact of increasing reconfiguration time. If the reconfiguration time were sufficiently low, it would be further possible to introduce additional reconfiguration points in the program besides the ones created by non-work actions.

6.3 Architectural Model

To analyze a program, Seshan and I extended the architectural model from Cabezas and Püschel [27]. This model takes in a program as a sequence of LLVM IR operations, which has similarities to assembly. The operations are in SSA form and therefore enable direct tracking of dependencies between operations without analyzing resources such as registers or spills through the stack. Using the dependence graph of the program, the model computes how the instructions can be scheduled on an architectural back end. Most operations have a fixed latency once they can be scheduled to a function unit; however, memory operations require the degree of memory locality computed using reuse distance [38]. The model simplifies several aspects of a computer architecture, such as instruction fetch and branch prediction.

For each operation (i.e., instruction), the architectural model computes whether the instruction fits into the current fetch bundle and then passes it on for scheduling. If there are no reorder buffers or reservation stations available, the instruction itself and instruction fetch are stalled accordingly. The instruction is then scheduled based on its dependencies, and then the next cycle where its appropriate function unit is available. When all instructions have been scheduled, the model can report information about the overlap of time spent on different resources and resource stalls. For the purposes here, the focus is on the number of cycles to schedule the operations provided.

This model has two limitations. First, it relied on the LLVM interpreter to provide the IR operations of a program's execution. The interpreter has its own set of limitations and slowdowns. Second, it is inherently sequential and only models one thread of execution. Rather than rewrite the model to support simulation of parallel execution, I elected to leverage the power of Contech's task graph and instead simulate the parallel aspects of a program separately.

6.3.1 Integration with Program Model

The first step to driving the comprehensive model with Contech task graphs is replacing the LLVM interpreter. Although it could, Contech does not store the parallel program's LLVM IR into the task graph. Instead, it is a simple task to take the final, instrumented IR file and reconstruct the original program, particularly as the instrumentation does not modify the original program (excepting certain OpenMP constructs), and instead injects the additional instructions required.

The Contech task graph contains sequences of basic blocks, along with memory accesses in each work task. To integrate with the program model, the tool rereads the program's LLVM IR and identifies the static operations that are associated with each basic block. The execution of a single context is replayed, again given the requirements of the original model, converting each basic block into the appropriate sequence of IR operations and providing the address or other information when the IR operation is accessing memory. Thereby, the Contech basic block sequence is transformed into the IR required by the architectural model.

Black box elements such as `malloc` are left as function calls. Should an analysis need the details of one of the black box functions, it is possible to establish a canonical version that can be supplied. In general, these calls constitute a small portion of the overall execution (see Section 4.2.1.1), as well as many calls are non-work operations such as lock acquires for which any detailed version would have to convey additional information such as the wait time.

6.3.2 Modeling Communication and Coherence

In order to model the communication between threads, I elected to reuse an existing task graph analysis rather than extend the sequential model. The objective is to quantify the communication of the task graph into one or more numerical quantities called *comm factors*, which would enable the sequential model to account for the

effects of communication. The task graph analysis classifies the reads and writes in the program into whether the memory operations communicated with another context, thus classifying any data movement between contexts as communication and not just operations from specific sharing patterns such as producer / consumer. It does so in effect by treating the reads and writes as if they operated in an MSI (or derivative) coherence protocol with caches of unlimited size, with a specific cache-line size for tracking. Each read adds itself to the list of sharers on that cache line, while each write will clear the list of sharers except itself. If the context making the request was not part of the list of sharers, then the access is communicating (based on the assumed cache-line granularity). The analysis runs using the the memory accesses of a task graph, one task at a time, following the standard breadth-first traversal of the task graph. This approach is similar to Barrow-Williams [13], with several key changes.

First, the memory accesses are viewed on a task by task basis, which may not detect some interleaved communication; however, as that communication would occur without an explicit ordering it would be a data race. To verify this assumption regarding data races, I used a separate analysis of the task graphs (based on the a prior data race detector [121]), which identifies finds the first occurrence of two accesses to same address that have no chain of task ordering dependencies between them. These accesses are unordered and therefore are races. Table 9 shows the quantity of races, bytes transferred and then bytes accessed. For races to be negligible, the quantity of races must be significantly less than the bytes transferred. Therefore from the analysis, I have concluded that as races are rare in the benchmarks (except for *canneal*), there is minimal inaccuracy from this modeling choice. Hence, by modeling the accesses at the task granularity, the graph's existing ordering constraints will guarantee that no amount of speedup or slowdown of any task will change the ordering

Table 9: Data Race Analysis Results (in Bytes)

benchmark	Race	Transfered	Accessed
barnes	633320	5.37E+06	9.76E+09
blackscholes	0	64	1.38E+08
bodytrack	60	8.43E+06	2.48E+09
canneal	11784	60	7.06E+09
cholesky	240	1.11E+07	2.10E+09
dedup	11140	3.26E+07	3.91E+08
ferret	184456	7.31E+06	6.76E+09
fft	4	6.29E+07	4.56E+09
fmm	207844	1.48E+07	1.27E+10
lu_cb	8	8.02E+06	8.77E+09
lu_ncb	8	4.21E+06	8.77E+09
ocean_cp	32	9.21E+06	1.60E+10
ocean_ncp	48	1.29E+08	1.58E+10
radiosity	84612	1.75E+07	7.66E+09
radix	16	1.28E+08	3.54E+09
raytrace	592	1.34E+06	2.65E+10
streamcluster	9	4.82E+06	6.92E+09
swaptions	0	64	8.73E+09
volrend	6256	2.67E+06	2.08E+09
water_spatial	16	2.98E+06	1.72E+10

of the tasks and therefore the measurement of (non-race) communication will be preserved across different architectures.

Second, as the task graph contains both `malloc` and `free` invocations, it can distinguish whether the communication involved the same data structure or a reallocation of the same memory location, where the latter is not actually communication intended by the program and instead an artifact of a particular execution and library implementation. As a consequence of these changes, the communication model gives an architecturally independent measurement of the inter-thread communication. The specific cost of the measured communication may be affected by architecture features, such as network on chip (NoC) designs and cache protocols.

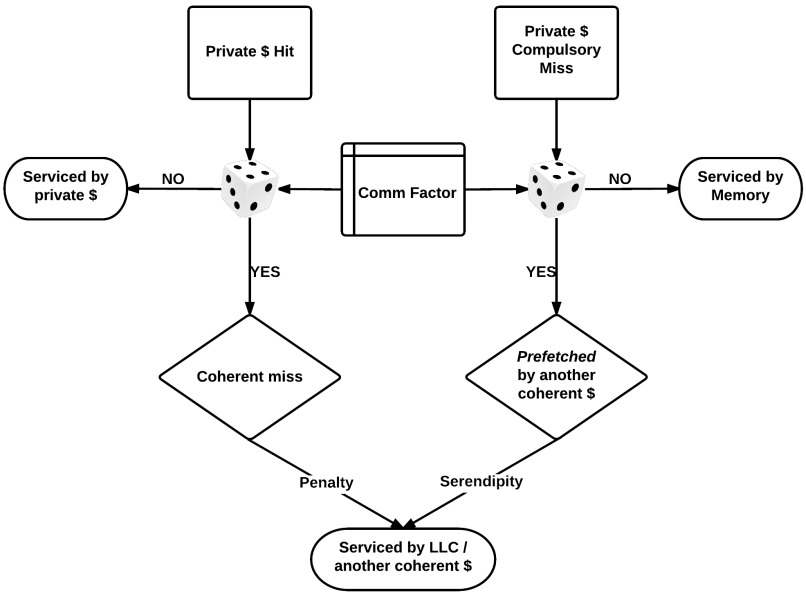


Figure 34: Simulating Communication with Randomness

Comm factor is defined as the ratio of number of memory accesses that require the data to be provided by another context to the total number of accesses. In other words:

$$comm\ factor = \frac{\#(communications)}{\#(accesses)}$$

In the extreme, a comm factor of 1.0 indicates that every request to any given line X would result in a communication or data transfer, indicating that some other processor invariably accessed X in the past. In general, comm factor gives the combined probability of either an access being a coherence miss (*penalty*) or of an access benefiting (*serendipity*) from another context having accessed the data already.

As a motivating example, consider two processors, P0 and P1, and a cache line X. Communicating accesses can be classified into two scenarios:

- Serendipity: Say P1 accesses X at t_0 and P0 accesses X for the first time at $t_1 | t_1 > t_0$. The single-threaded reuse distance for P0 (RD_0) predicts a compulsory miss at t_1. However, P0 does not need to fetch X all the way from long latency main memory as it can be serviced from P1 or the LLC.

- Penalty: Say P0 accesses X at t_0 and P1 writes X at $t_1 | t_1 > t_0$ and P0 accesses X again at $t_2 | t_2 > t_1$. Irrespective of what RD_0 predicts at t_2, we must model communication of data from LLC to P0.

In both cases, the latest access by P0 is recorded as a communication. However, subsequent P0 accesses to X do not result in on-chip data communication, as long as there is no interleaving P1 access. As a first order model, we simulate such on-chip communication as hits in the LLC.

Using the percentage of communicating reads and writes, the detailed model can then treat the same percentage of memory operations as communicating. Figure 34 shows how the model will probabilistically adjust cache hits and misses based on the *comm factor*. On each memory operation, the model takes the statistical chance (*comm factor*) that the operation would not have its single-threaded reuse distance, but instead require accessing a different level of the memory hierarchy. Such an operation is equivalent to there being a communicating operation from a different thread within the modeled thread's reuse distance.

The comm factor can be computed on either a per-task or per-context basis. Unless otherwise noted, all results use the per-task comm factor for better accuracy.

6.4 Parallel Program Modeling

The serial architectural model calculates the approximate execution rate of the provided set of instructions. By providing the instructions from entire tasks in the task graph, we can compute an approximate execution rate for that task. Using different configurations, the model will compute different execution rates, which give estimated speedups for that task's execution on the different microarchitectures. However, rate-based speedups of parallel programs can be inaccurate, when their computation includes inter-thread related instructions, such as waiting on a lock, without modeling the actual timing of the other threads.

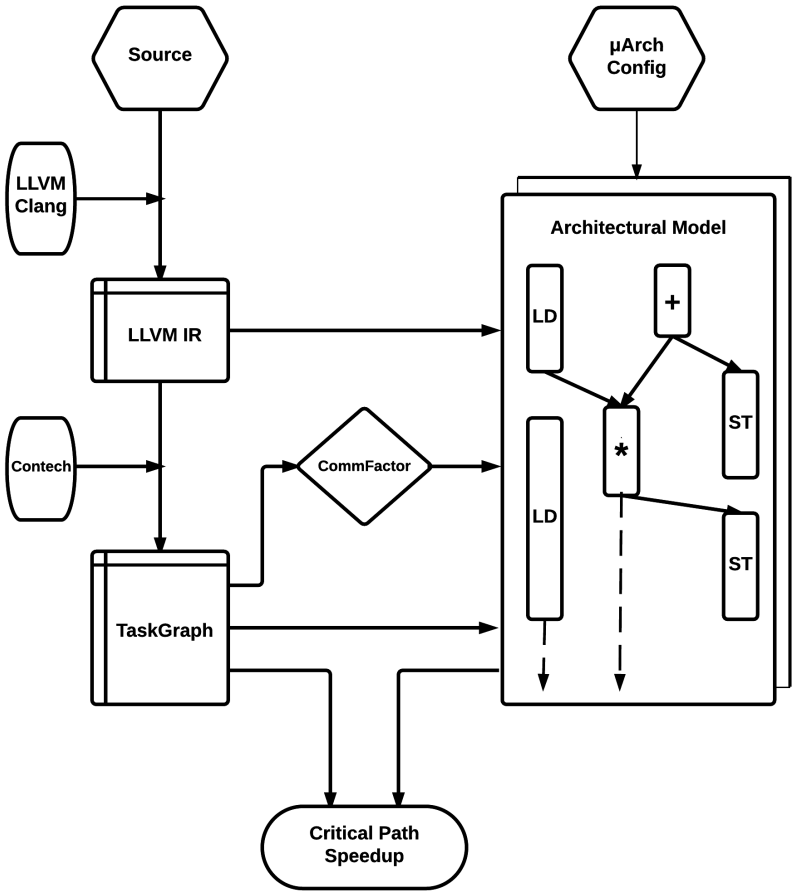


Figure 35: Using task graphs to drive parallel program modeling and analysis

Using the task graph from Contech, the parallel program analysis respects the program's order, separates out the non-accelerated components (e.g., synchronization), and computes a whole program speedup from the rate-based improvements of different tasks. By respecting the order and separating out synchronization and other operations, the rate-based improvement represents an improvement in that task's execution time and the overall program order then reveals whether this improvement impacts execution time. This analysis is still limited in that it is preserving the original ordering of tasks, for example if the speed up of one task is sufficiently large versus another, that task might be in place to acquire a lock in a different order than the original execution. The workflow of this combined analysis is shown in Figure 35.

The task graph is limited in that it only records the parallelism exposed by the program's execution. For this model, I am not modeling any scheduling of tasks beyond their dependencies, rather making the idealized assumption that there are always enough cores available to execute the ready tasks of the program. Introducing a scheduling aspect would add additional complexities to understanding the benefits of this system.

6.5 Experimental Results

ALGORITHM 2: Critical Path

```
input  : Task Graph of Program
input  : Speedups of Each Task by Configuration
output: Length of Critical Path
for t ∈ TaskGraph do
    /* Not shown:  Length of non-work tasks must account for lock
       contention and other effects                                  */
    speedup ←GetMaxSpeedup (t);
    length ← t.getEndTime () - t.getStartTime ();
    for p ∈ t.getPredecessorTasks do
        if PathLengthAt (p) > maxPath then
            MaxPath ← PathLengthAt (p);
        end
    end
    MaxPath ← MaxPath + length/speedup;
    PathLengthAt (t)← MaxPath;
end
return PathLengthAt (ROI_END);
```

I used Algorithm 2 to find the length of the critical path for a set of possible modeled configurations (i.e., increasing number of function units, buffer sizes, et cetera). There are two interesting additions. First, the algorithm must account for the actual time contribution of non-work tasks. For example, a sync task acquiring a lock is ordered after the release task for that lock. The task graph model has no explicit understanding of contention, so the acquire sync task may include the time waiting for the lock to be released. Also in many workloads, there is a master thread creating and joining with all of the worker threads. Therefore the first join task has

Table 10: Benchmark Characteristics

Benchmark	Modeled ROI Length Accuracy	Total Modeled Ops (Million)
barnes	99.8%	1794
blackscholes	97.9%	109
bodytrack	99.7%	2473
canneal	100.0%	670
cholesky	99.7%	656
dedup	99.8%	1362
ferret	100.0%	2671
fft	99.9%	1945
fmm	99.8%	6818
lu_cb	95.9%	1786
lu_ncb	97.9%	1791
ocean_cp	95.6%	4468
ocean_ncp	97.1%	4434
radiosity	100.0%	4362
radix	99.4%	2007
raytrace	100.0%	6545
streamcluster	85.5%	5729
swaptions	99.9%	2698
volrend	94.7%	3224
water_spatial	99.6%	4423

a significant time gap between the start of the join (shortly after the last create) and its end (at the end of the workload). Second, the duration of any work task may be scaled by its speedup, which is also complicated when the decision to reconfigure must also account for the cost of doing so. The full implementation of the Critical Path analysis can also report additional information such as the complete path through the graph and which configuration is used by each task.

Each benchmark is run through the model for the entire region of interest (ROI), if supplied by the benchmark suite, otherwise for the entire program. Table 10 shows the modeled ROI length and the size of program in LLVM IR operations. The benchmarks average 3 billion operations in their ROIs, and the model executes at approximately 0.5 MoPS (millions of IR operations per second), which averages a 1200x slowdown from the native execution speed. To compute the accuracy of the model's ROI, it takes

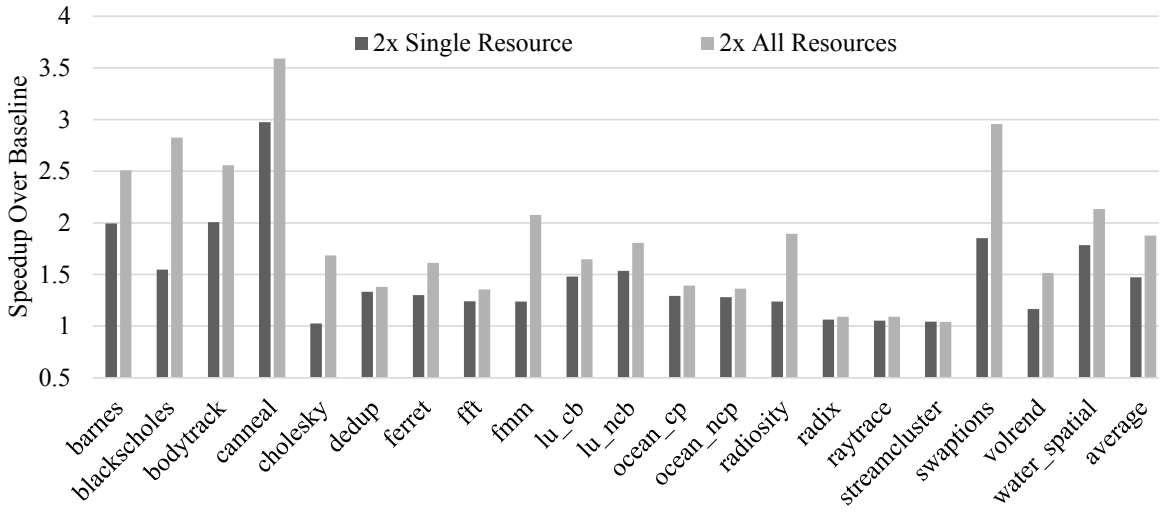


Figure 36: Speedups of ROI in PARSEC Benchmarks

the start and end times for the ROI and then finds the critical path when there is no speedup. This verifies whether the critical path algorithm has accurately determined the time spent waiting for locks, barriers, and other non-work operations that may exist along the critical path. Finding the contribution of each non-work operation is non-trivial, as the operations are treated as black boxes, such that the internal breakdown of different timing components are not recorded. An additional model of synchronization would break the synchronization task's time into time waiting and time transferring. The time waiting can change based on the execution speed of the ancestor task, while the transfer time would remain independent and instead reflects the design of the underlying component (mutex, atomic, et cetera).

For most workloads, the critical path model covers a high percentage of the ROI. *Streamcluster* is one exception, as on the test machine, the threads created in *streamcluster* did not begin executing for several million cycles, possibly as it spawns more threads (32) than there are cores (24), which introduces a delay on the critical path for which the model does not account.

The task graphs used in the following experiments were collected on 24-threaded x86 processors; however, the task graph and architectural model are ISA-independent

Table 11: Number of Operations Concurrently Issued in Blackscholes Critical Task (1:0)

Degree of	Floating Point			L1	
Concurrent Issue	ADD	MUL	DIV	LD	ST
1	527616	419506	351241	51287	101638
2	955360	478586	57374	102046	24
3	342657	768306	153	152610	0
4	264100	912492	772	305936	0
5	100075	685920	60	0	0

and could model programs executed on other ISAs. Modeling different core counts is more difficult, although it is possible when the program is oversubscribed. For example, *ferret* spawns 64 threads to execute its pipeline, requiring a higher core count from the model.

6.5.1 Performance Improvement

Figure 36 provides the speedup versus the baseline simulated from applying the per-task speedups for the different combinations of hardware configurations. Each configuration doubles the quantity of one of the resources over the baseline presented in Table 7, which results in the number of units increasing from 1 to 2 except for integer ALUs that increase from 3 to 6. All configurations also double the size of the buffers. The 2X Single Resource bar provides the speedup from Algorithm 2. The 2X All Resources introduces a new configuration where all of the resources in Table 7 have been doubled.

The difference between the two bars demonstrates the degree to which each task is constrained to a single resource versus benefiting from increasing other resource types. *Barnes*, *canneal*, and *water_spatial* are benchmarks where the critical tasks are strongly coupled to a single resource type. In contrast, *fmm* and *radiosity* show minimal gains unless all of the resources are increased, which suggest their tasks require a diversity of resources.

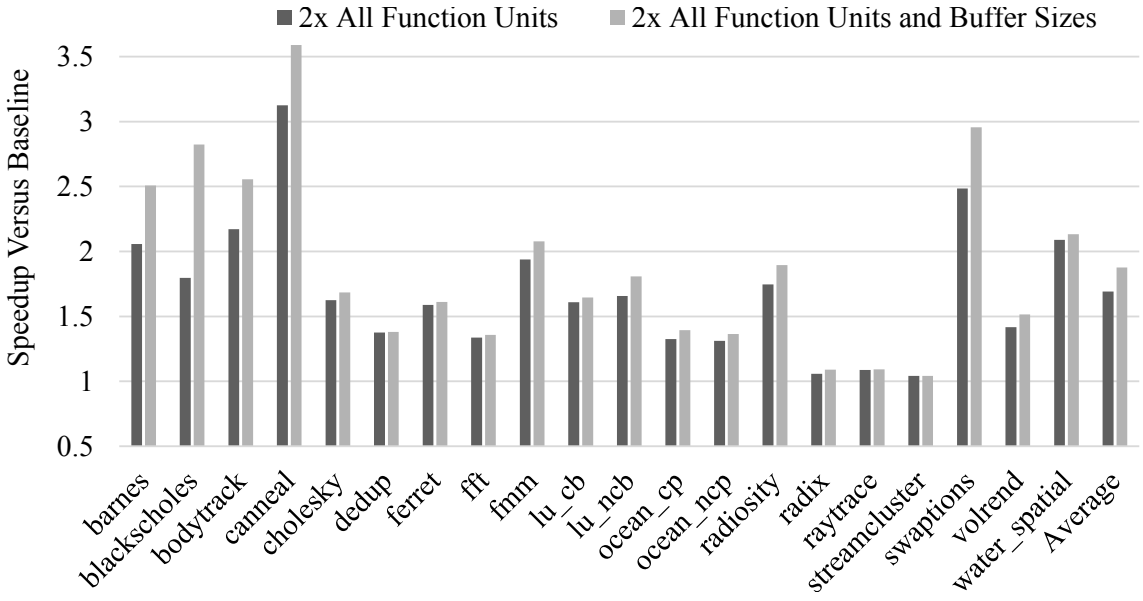


Figure 37: Speedups of ROI in PARSEC Benchmarks Accounting for Buffer Increases

To further understand the speedups, the resource usage of each type of execution unit was measured when unlimited execution units were available, which provides a histogram of the number of units needed by each task in the program. Table 11 presents one example task from *blackscholes*. Each value is the number of operations concurrently issued to that quantity of FUs. For instance, there were almost 1 million floating point adds issued without stall when 1 other floating point add was issued, for a concurrent issue of 2. For this critical task in *blackscholes*, the critical path analysis indicated that introducing an additional divider provides the greatest increase in performance at over 50% (2X Single Resource). After which, the performance is constrained by multipliers and so increasing quantity of the other FUs further improves performance to 2.8x speedup (2X All Resources).

Increasing the size of the instruction buffers can lead to improved performance from the larger scheduling window, usually proportional to the log of the size, and assuming there are sufficient execution resources. Figure 37 shows the performance difference from enabling all of the additional execution resources and only changing whether the sizes of the buffers are increased.

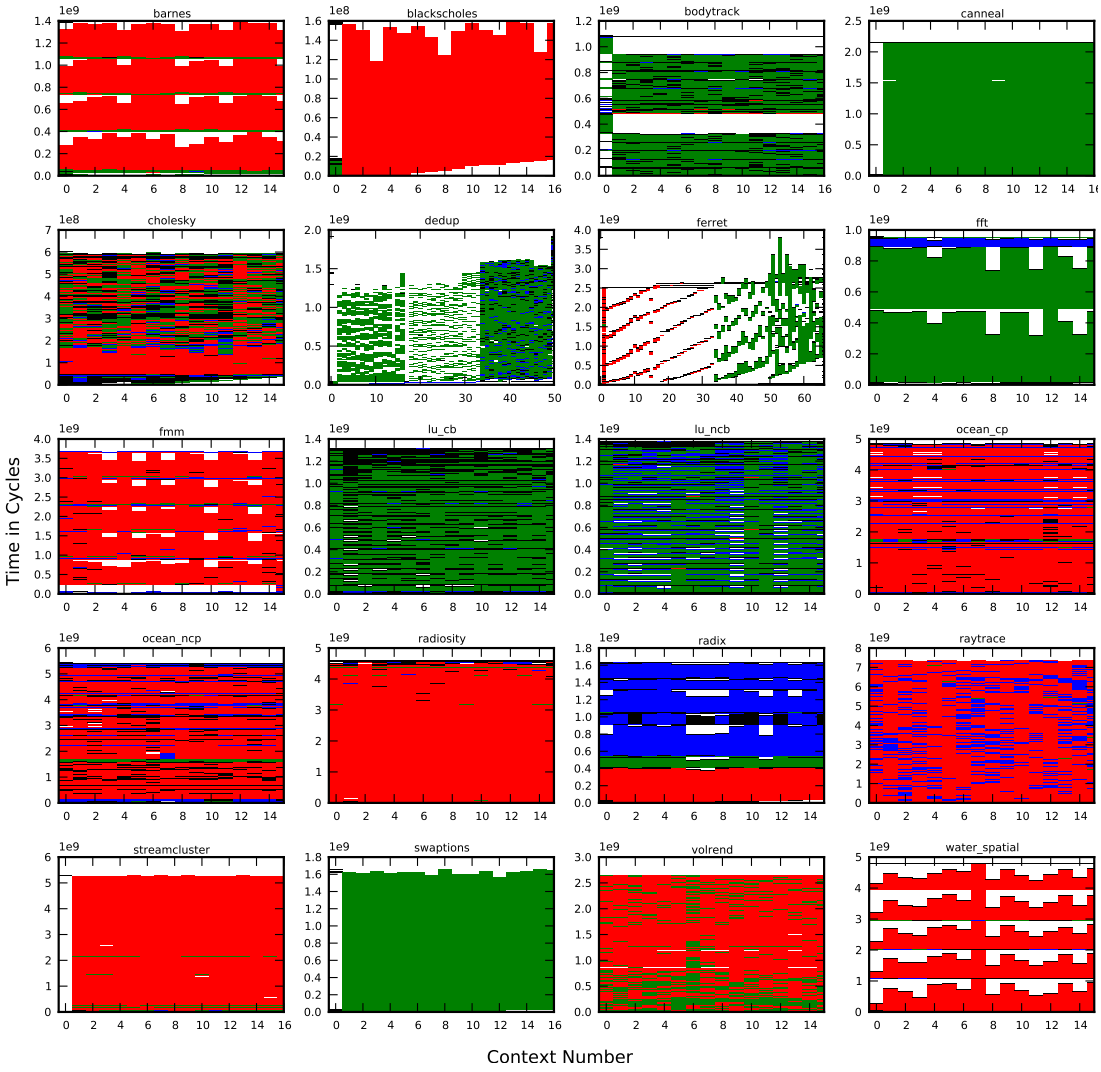


Figure 38: Configuration Type Used in Accelerating the ROI in PARSEC Benchmarks (Green - Integer, Red - Floating Point, Blue - Memory, Black - Minimal Speedup)

6.5.2 Context Phase Variation

Figure 38 plots the type of the configuration from the 2X single resource increase applied to each task in the ROI. The types are sorted into five categories: integer (green), floating point (red), memory (blue), none / minimal speedup (black), and non-work (white). It is important to understand that even regions of the same type, integer for instance, may be benefiting from different integer resources. The white regions of time give us insight into the program's parallelism over time. For many

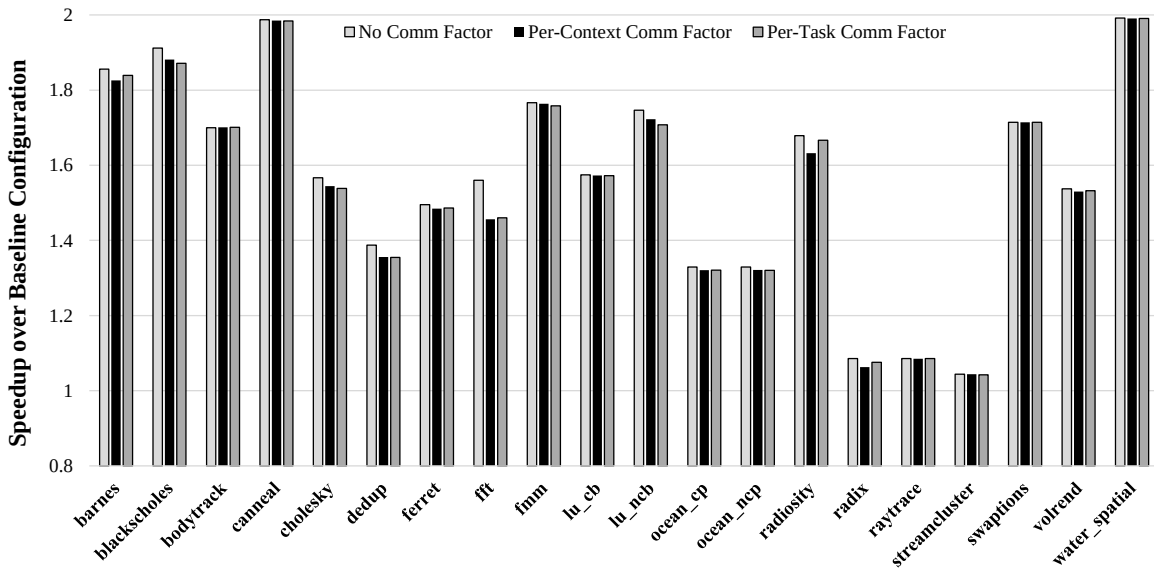


Figure 39: Speedups from Varying Comm Factor Granularity in PARSEC Benchmarks (2X All Configuration) Using Earlier Architectural Model

workloads, the white regions reveals the barrier nature (c.f., *water_spatial*) or the pipeline pattern used within *dedup* and *ferret*. *Blackscholes* and *swaptions* are task parallel and therefore have no synchronization to introduce task boundaries that would enable different configurations to be modeled and selected. *Canneal* is similar, although it has several synchronization tasks that provide opportunities for possible dynamic reconfiguration. *Cholesky* and *Raytrace* are two examples of workloads that vary between different types as the tasks execute discrete work.

6.5.3 Comm Factor Granularity

In introducing the comm factor component to parallel program modeling, I explored two basic questions. What benefit or impact does it have to the program? And should the granularity of comm factor be accounted by task or by context? Using an earlier iteration of the model, the program speedups were measured using three variations on *comm factors*: none, per-context, and per-task. The results are plotted in Figure 39 and show that while running without a comm factor has the best speedup, the results are mixed as to whether the per-context or per-task comm factors impact

the performance more. *Canneal* is one of several benchmarks that shows minimal difference in speedups when *comm factors* are used in the model. This leads back to Table 9, where *dedup* has the highest relative bytes of communication and thus the use of *comm factors* predicts a slower execution. *Barnes* and *radiosity* both have lower speedups with per-context over per-task *comm factors*. This contrasts with *lu_ncb*, where the per-task *comm factor* experiment has the least speedup. These results generally indicate that penalty is more prevalent than serendipity, such that model estimates that other contexts are requesting the cache lines more frequently than cache misses becoming cache hits.

6.5.4 Staticness of Configurations

In order to apply the reconfigurations into a compiled program, the system needs to identify specific points in the binary as reconfiguration points, as well as determine the best configuration for this point across all executions. I looked at the ideal configuration selected for each task, as well as the final portion of the path leading into that task. The last basic block before a task and the first into the new task establishes a particular point in the program. If these points have similar configurations, then the configuration can be applied at that transition rather than reliant on dynamic information. Table 12 shows the fraction of every transition A $\rightarrow$B, where each B would use the same dynamic configuration, as well as the percentage of the dynamic reconfiguration speedup achieved when using static reconfigurations. To determine the static configuration for each point, the critical path analysis evaluated each configuration and computed the aggregate speedup each would provide for a given location. Then using the best per-task static configuration, the speedups were again measured. All workloads achieved better than 90% of the dynamic configuration speedups, which affirms the validity of applying a specific resource configuration to a code region. Even workloads that have less stability of configurations for a code region, the tasks

Table 12: Staticness of Ideal Per-Task Configurations

Benchmark	Fraction of Configurations for Code Section	% Dynamic Speedup Achieved with Static Configuration
barnes	0.88	100.0%
blackscholes	1.00	100.0%
bodytrack	0.88	99.5%
canneal	1.00	100.0%
cholesky	0.78	99.8%
dedup	0.88	99.6%
ferret	0.67	97.2%
fft	1.00	100.0%
fmm	0.89	91.7%
lu_cb	0.85	100.0%
lu_ncb	0.74	99.3%
ocean_cp	0.93	99.9%
ocean_ncp	0.91	99.1%
radiosity	0.87	99.5%
radix	0.80	99.1%
raytrace	0.61	99.2%
streamcluster	1.00	100.0%
swaptions	1.00	100.0%
volrend	0.70	97.2%
water_spatial	0.89	100.0%

seeing better speedup than the static configuration are either off of the critical path or, more likely, receiving near-optimal speedups from other configurations.

In this work, I have not explored how the program would communicate the reconfiguration information. Generally, such information would be communicated to the hardware using an ISA extension. Vandierendonck and De Bosschere [119] showed how certain information could be passed to the compiler within the existing ISA by creating an implicit ISA of selecting a specific register out of an equivalent set has semantic meaning. Reconfiguration is far more costly to mistake than their example of branch mispredictions; however, this suggests other options than potentially breaking compatibility through ISA changes.

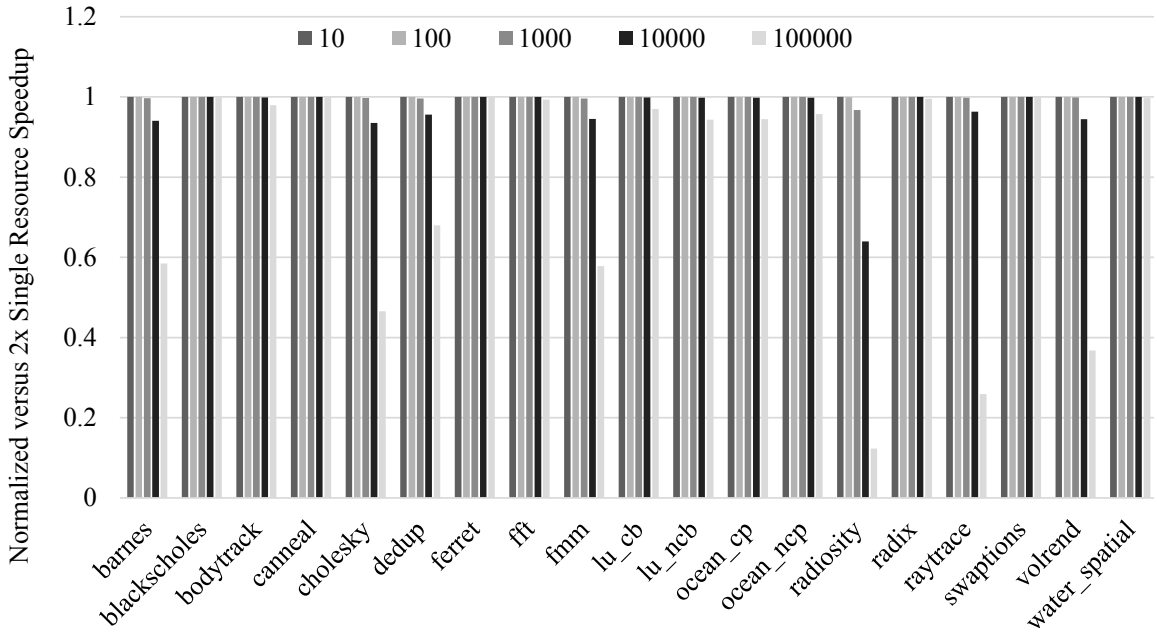


Figure 40: Impact of Reconfiguration Latency (in cycles) on Speedup

6.5.5 Overlapping Reconfiguration and Wait Times

In so far as it has been established that static reconfiguration decisions would benefit a parallel program, the next question is what is the impact of the reconfiguration time. It must be noted that the extended architectural model flushes the pipeline (but maintains cache re-use distance) at each task boundary, so as to not overestimate available ILP should a reconfiguration be actually triggered. The pipeline would naturally be flushed as part of executing most non-work operations, by for example, the atomic operations executed as part of acquiring a lock.

Prior work [40] [58] [104] [113] has shown that flushing the pipeline, resizing OoO buffers and powering up/down functional units takes a few hundred cycles at most. However, as this is specific to an implementation and a technology node, and that this framework may be extended to support more complex reconfigurations, the reconfiguration latency is varied in order to evaluate its impact on speedup. Figure 40 summarizes this impact by plotting that ratio of 2x oracle speedup obtained with arbitrary reconfiguration latencies (in cycles) to the 2x oracle speedup obtained with

0 reconfiguration latency. It can be seen that the 2x oracle speedup is resilient to reconfiguration latency for up to at least a few thousand cycles.

In generating Figure 40, any potential dynamic reconfiguration needed for a work task is overlapped with time taken by a non-work task that immediately precedes the work task. If the reconfiguration time exceeds the available *wait* time in the corresponding context, the work task that prefers a new configuration, is inevitably delayed. These correspond to the terms *overlap* and *overhead*, as was explained in Section 6.2.1, Figure 33. A further analysis may consider the impact of optionally not performing reconfiguration in the presence of *overhead*, but as long as the latency of reconfiguration is limited to a few thousand cycles, as is the case here, this is apparently not necessary, as evidenced by the initial resilience to reconfiguration latency in Figure 40.

Another insight gained from this sensitivity analysis is that dynamic reconfiguration is likely to yield better speedup than migrating critical sections to specialized cores. As thread migration often involves flushing out cache entries, the overhead can potentially reach the order of a million cycles [112]. This is not to say that thread migration is an infeasible solution as one can always come up with smart mapping and scheduling policies to ensure such pathological cases do not occur. The overhead itself is relatively lower by employing dynamic reconfiguration, and we have the added advantage of hiding this overhead by overlapping it with task synchronization.

6.6 Summary

This chapter presented a novel reconfigurable architecture, whereby the compiler works with a Contech-based analysis tool to determine the best resources to additionally activate for each executing task. This analysis required using all aspects of the Contech task graph representation, and consequently helped find several bugs in

the instrumentation. The speedup model showed that small increases in architectural resources can significantly accelerate parallel programs, averaging 40% across programs from the PARSEC benchmark suite.

CHAPTER 7

CONCLUSION

As hardware changes and programs grow increasingly more complex, computer architects and programs need improved support from simulators and tools. Current approaches have been targeted to specific problems and the fields are without unified frameworks to represent and analyze modern programs. Developing new analyses requires identifying the data required, preparing the appropriate instrumentation, and then significant effort to ensure that this instrumentation does not disrupt the phenomena being measured.

This dissertation establishes, in Chapter 2, Contech's task graph representation as capable of supporting a diversity of parallel programs across both different languages and parallel paradigms. The Contech framework, as described in Chapter 4, provides high performance instrumentation for collecting a task graph from a program's execution with minimal performance perturbation. In Chapter 5, the framework's support for program analysis was discussed and then demonstrated with a diversity of backends. This demonstration culminated in Chapter 6 where an extensive analysis of a reconfigurable architecture was modeled and driven using Contech, thereby exercising the complete task graph.

Moving forward, the Contech framework can be further improved and extended. For example, the instrumentation support can be added to other languages, and the support for runtime paradigms will be extended as new versions provide additional features and parallel constructs. The instrumentation itself can still achieve further

performance improvements, such as addressing frequent creations in *lulesh* or improving the buffer overflow heuristics. The value of Contech's high performance instrumentation would be further established by studying the impact of instrumentation (i.e., the probe effect) in parallel programs. Finally, the work on the parallel program modeling in Chapter 6 is continuing, especially as the model should be validated by a separate simulator.

In summary, Contech provides a framework to collect a rich parallel program representation across a diversity of languages and paradigms and do so with a low overhead from the high performance instrumentation, along with APIs for constructing program analyses utilizing Contech's task graph representation.

REFERENCES

[1] Adve, V. S. and Sakellariou, R., "Compiler synthesis of task graphs for parallel program performance prediction," in *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, LCPC '00, (London, UK, UK), pp. 208–226, Springer-Verlag, 2001.

[2] Adve, V. S. and Vernon, M. K., "Parallel program performance prediction using deterministic task graph analysis," *ACM Trans. Comput. Syst.*, vol. 22, pp. 94–136, Feb. 2004.

[3] Agrawal, K., Leiserson, C., and Sukha, J., "Executing task graphs using work-stealing," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 1–12, April 2010.

[4] Almeida, V. A. F., Vasconcelos, I. M. M., Árabe, J. N. C., and Menascé, D. A., "Using random task graphs to investigate the potential benefits of heterogeneity in parallel systems," in *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, (Los Alamitos, CA, USA), pp. 683–691, IEEE Computer Society, 1992.

[5] Ansaloni, D., Binder, W., Heydarnoori, A., and Chen, L. Y., "Deferred methods: Accelerating dynamic program analysis on multicores," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, (New York, NY, USA), pp. 242–251, ACM, 2012.

[6] Arnold, M. and Ryder, B. G., "A framework for reducing the cost of instrumented code," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, (New York, NY, USA), pp. 168–179, ACM, 2001.

[7] Atachiants, R., Gregg, D., Jarvis, K., and Doherty, G., "Design considerations for parallel performance tools," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, (New York, NY, USA), pp. 2501–2510, ACM, 2014.

[8] Ayguadé, E., Badia, R. M., Jiménez, D., Herrero, J. R., Labarta, J., Subotic, V., and Utrera, G., "Tareador: a tool to unveil parallelization strategies at undergraduate level," pp. 1–8, 2015.

[9] Bach, M., Charney, M., Cohn, R., Demikhovsky, E., Devor, T., Hazelwood, K., Jaleel, A., Luk, C.-K., Lyons, G., Patil, H., and Tal, A., "Analyzing parallel programs with pin," *Computer*, vol. 43, no. 3, pp. 34–41, 2010.

[10] Ball, T. and Larus, J. R., "Efficient path profiling," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, (Washington, DC, USA), pp. 46–57, IEEE Computer Society, 1996.

[11] Barik, R., Zhao, J., and Sarkar, V., "Interprocedural strength reduction of critical sections in explicitly-parallel programs," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, PACT '13, (Piscataway, NJ, USA), pp. 29–40, IEEE Press, 2013.

[12] Barr, K., Pan, H., Zhang, M., and Asanovic, K., "Accelerating multiprocessor simulation with a memory timestamp record," in *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pp. 66–77, March 2005.

[13] Barrow-Williams, N., Fensch, C., and Moore, S., "A communication characterisation of splash-2 and parsec," in *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.*, pp. 86–97, 2009.

[14] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A., "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 66:1–66:11, IEEE Computer Society Press, 2012.

[15] Beckmann, C. J. and Polychronopoulos, C. D., "Microarchitecture support for dynamic scheduling of acyclic task graphs," in *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, (Los Alamitos, CA, USA), pp. 140–148, IEEE Computer Society, 1992.

[16] Bellard, F., "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), pp. 41–41, USENIX Association, 2005.

[17] Beu, J. G., Poovey, J. A., Hein, E. R., and Conte, T. M., "High-speed formal verification of heterogeneous coherence hierarchies," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, (Washington, DC, USA), pp. 566–577, IEEE Computer Society, 2013.

[18] Beu, J. G., Rosier, M. C., and Conte, T. M., "Manager-client pairing: A framework for implementing coherence hierarchies," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 226–236, ACM, 2011.

[19] Bhadauria, M., Weaver, V., and McKee, S., "Understanding parsec performance on contemporary cmps," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 98–107, Oct 2009.

[20] Bienia, C., *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[21] Blelloch, G. E., Gibbons, P. B., Matias, Y., and Narlikar, G. J., "Space-efficient scheduling of parallelism with synchronization variables," in *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, (New York, NY, USA), pp. 12–23, ACM, 1997.

[22] Blumofe, R. D. and Leiserson, C. E., "Space-efficient scheduling of multithreaded computations," in *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, (New York, NY, USA), pp. 362–371, ACM, 1993.

[23] Bruening, D., Duesterwald, E., and Amarasinghe, S., "Design and implementation of a dynamic optimization framework for windows," in *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4*, 2000.

[24] Bruening, D., Garnett, T., and Amarasinghe, S., "An infrastructure for adaptive dynamic optimization," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, (Washington, DC, USA), pp. 265–275, IEEE Computer Society, 2003.

[25] Bruening, D., Zhao, Q., and Amarasinghe, S., "Transparent dynamic instrumentation," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, (New York, NY, USA), pp. 133–144, ACM, 2012.

[26] Bryan, P., Poovey, J., Beu, J., and Conte, T., "Accelerating multithreaded application simulation through barrier-interval time-parallelism," in *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pp. 117–126, Aug 2012.

[27] Cabezas, V. and Puschel, M., "Extending the roofline model: Bottleneck analysis with microarchitectural constraints," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pp. 222–231, Oct 2014.

[28] Carlson, T., Heirman, W., Van Craeynest, K., and Eeckhout, L., "Barrierpoint: Sampled simulation of multi-threaded applications," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 2–12, March 2014.

[29] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V., "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems,*

Languages, and Applications, OOPSLA '05, (New York, NY, USA), pp. 519–538, ACM, 2005.

[30] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.-H., and Skadron, K., "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, Oct 2009.

[31] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K., "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, (Washington, DC, USA), pp. 44–54, IEEE Computer Society, 2009.

[32] Che, S., Sheaffer, J., Boyer, M., Szafaryn, L., Wang, L., and Skadron, K., "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pp. 1–11, Dec 2010.

[33] Criminisi, A., Shotton, J., and Konukoglu, E., "Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning," *Foundations and Trends in Computer Graphics and Vision: Vol. 7: No 2-3, pp 81-227*, 2012.

[34] Cui, H., Wu, J., Gallagher, J., Guo, H., and Yang, J., "Efficient deterministic multithreading through schedule relaxation," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 337–351, ACM, 2011.

[35] Culler, D. E., Karp, R. M., Patterson, D., Sahay, A., Santos, E. E., Schauser, K. E., Subramonian, R., and von Eicken, T., "Logp: A practical model of parallel computation," *Commun. ACM*, vol. 39, pp. 78–85, Nov. 1996.

[36] Czechowski, K., Battaglino, C., McClanahan, C., Chandramowlishwaran, A., and Vuduc, R., "Balance principles for algorithm-architecture co-design," in *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, (Berkeley, CA, USA), pp. 9–9, USENIX Association, 2011.

[37] Deniz, E., Sen, A., Kahne, B., and Holt, J., "Minime: Pattern-aware multicore benchmark synthesizer," *Computers, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2014.

[38] Ding, C. and Zhong, Y., "Predicting whole-program locality through reuse distance analysis," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, (New York, NY, USA), pp. 245–257, ACM, 2003.

[39] Dolan-Gavitt, B. F., Hodosh, J., Hulin, P., Leek, T., and Whelan, R., "Repeatable reverse engineering for the greater good with panda," Columbia University Academic Commons, 2014.

[40] Dubach, C., Jones, T. M., Bonilla, E. V., and O'Boyle, M. F. P., "A predictive model for dynamic microarchitectural adaptivity control," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 485–496, IEEE Computer Society, 2010.

[41] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D., "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.

[42] Etsion, Y., Cabarcas, F., Rico, A., Ramirez, A., Badia, R. M., Ayguade, E., Labarta, J., and Valero, M., "Task superscalar: An out-of-order task pipeline," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 89–100, IEEE Computer Society, 2010.

[43] Fog, A., "Instruction tables." Available at www.agner.org/optimize/instruction_tables.pdf as on May 21, 2015.

[44] Fog, A., *Instruction Tables*. Technical University of Denmark, December 2014. Available at http://www.agner.org/-optimize/instruction_tables.pdf.

[45] Froyd, N., Mellor-Crummey, J., and Fowler, R., "Low-overhead call path profiling of unmodified, optimized code," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, (New York, NY, USA), pp. 81–90, ACM, 2005.

[46] Gerasoulis, A., Venugopal, S., and Yang, T., "Clustering task graphs for message passing architectures," in *Proceedings of the 4th international conference on Supercomputing*, ICS '90, (New York, NY, USA), pp. 447–456, ACM, 1990.

[47] Goel, A., Roychoudhury, A., and Mitra, T., "Compactly representing parallel program executions," in *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, (New York, NY, USA), pp. 191–202, ACM, 2003.

[48] Govindaraju, V., Ho, C.-H., and Sankaralingam, K., "Dynamically specialized datapaths for energy efficient computing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 503–514, Feb 2011.

[49] Graham, S. L., Kessler, P. B., and Mckusick, M. K., "Gprof: A call graph execution profiler," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, (New York, NY, USA), pp. 120–126, ACM, 1982.

[50] Greenhalgh, P., "Big. little processing with arm cortex-a15 & cortex-a7," *ARM White paper*, 2011.

[51] Guha, A., Zhang, Y., ur Rasool, R., and Chien, A. A., "Systematic evaluation of workload clustering for extremely energy-efficient architectures," *SIGARCH Comput. Archit. News*, vol. 41, pp. 22–29, May 2013.

[52] Gupta, G. and Sohi, G. S., "Dataflow execution of sequential imperative programs on multicore architectures," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 59–70, ACM, 2011.

[53] Ha, J., Arnold, M., Blackburn, S. M., and McKinley, K. S., "A concurrent dynamic analysis framework for multicore hardware," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, (New York, NY, USA), pp. 155–174, ACM, 2009.

[54] Hawkins, B., Demsky, B., Bruening, D., and Zhao, Q., "Optimizing binary translation of dynamically generated code," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, (Washington, DC, USA), pp. 68–78, IEEE Computer Society, 2015.

[55] He, Y., Leiserson, C. E., and Leiserson, W. M., "The cilkview scalability analyzer," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, (New York, NY, USA), pp. 145–156, ACM, 2010.

[56] Heumann, S. T., Adve, V. S., and Wang, S., "The tasks with effects model for safe concurrency," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, (New York, NY, USA), pp. 239–250, ACM, 2013.

[57] Hower, D. R. and Hill, M. D., "Rerun: Exploiting episodes for lightweight memory race recording," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, (Washington, DC, USA), pp. 265–276, IEEE Computer Society, 2008.

[58] Hu, S., Valluri, M., and John, L. K., "Effective management of multiple configurable units using dynamic optimization," *ACM Trans. Archit. Code Optim.*, vol. 3, pp. 477–501, Dec. 2006.

[59] Ibtesham, D., Arnold, D., Bridges, P., Ferreira, K., and Brightwell, R., "On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance," in *Parallel Processing (ICPP), 2012 41st International Conference on*, pp. 148–157, Sept 2012.

[60] Intel Corporation, *Cilk Plus/LLVM*, July 2015. http://cilkplus.github.io/.

[61] Intel Corporation, *OpenMP/LLVM*, July 2015. http://clang-omp.github.io/.

[62] Intel Corporation, Santa Clara, CA, *Intel 64 and IA-32 Architectures Software Developer Manuals*, 2014. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[63] ISO/IEC 9899:2011 WG14, *Programming Languages - C (C11)*, April 2011.

[64] Iyer, B. V., *Length adaptive processors: a solution for the energy/performance dilemma in embedded systems*. ProQuest, 2009.

[65] Jee, K., Kemerlis, V. P., Keromytis, A. D., and Portokalidis, G., "Shadowreplica: Efficient parallelization of dynamic data flow tracking," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, (New York, NY, USA), pp. 235–246, ACM, 2013.

[66] Jin, H., Frumkin, M., and Yan, J., "The openmp implementation of nas parallel benchmarks and its performance," Tech. Rep. NAS-99-011, NAS, October 1999.

[67] Jones, T. M., O'Boyle, M. F. P., Abella, J., and Gonzalez, A., "Software directed issue queue power reduction," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, (Washington, DC, USA), pp. 144–153, IEEE Computer Society, 2005.

[68] Jones, T. M., O'Boyle, M. F. P., Abella, J., González, A., and Ergin, O., "Exploring the limits of early register release: Exploiting compiler analysis," *ACM Trans. Archit. Code Optim.*, vol. 6, pp. 12:1–12:30, Oct. 2009.

[69] Jung, C., Rus, S., Railing, B. P., Clark, N., and Pande, S., "Brainy: Effective selection of data structures," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 86–97, ACM, 2011.

[70] Kambadur, M., Tang, K., and Kim, M. A., "Harmony: collection and analysis of parallel block vectors," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, (Washington, DC, USA), pp. 452–463, IEEE Computer Society, 2012.

[71] Kanev, S. and Cohn, R., "Portable trace compression through instruction interpretation," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pp. 107–116, April 2011.

[72] Karlin, I., Keasler, J., and Neely, R., "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[73] Kasikci, B., Zamfir, C., and Candea, G., "Automated classification of data races under both strong and weak memory models," *ACM Trans. Program. Lang. Syst.*, vol. 37, pp. 8:1–8:44, May 2015.

[74] Knobe, K., "Ease of use with concurrent collections (cnc)," in *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, (Berkeley, CA, USA), pp. 17–17, USENIX Association, 2009.

[75] Kulkarni, M., Burtscher, M., Inkulu, R., Pingali, K., and Casçaval, C., "How much parallelism is there in irregular applications?," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, (New York, NY, USA), pp. 3–14, ACM, 2009.

[76] Kumar, S., Hughes, C. J., and Nguyen, A., "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, (New York, NY, USA), pp. 162–173, ACM, 2007.

[77] Larus, J. R., "Abstract execution: A technique for efficiently tracing programs," *Softw. Pract. Exper.*, vol. 20, pp. 1241–1258, Nov. 1990.

[78] Larus, J. R., "Whole program paths," in *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, (New York, NY, USA), pp. 259–269, ACM, 1999.

[79] Lattner, C. and Adve, V., "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[80] Laurenzano, M. A., Peraza, J., Carrington, L., Tiwari, A., Ward, W. A., and Campbell, R., "A static binary instrumentation threading model for fast memory trace collection," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, (Washington, DC, USA), pp. 741–745, IEEE Computer Society, 2012.

[81] Lauterbach, G., "Accelerating architectural simulation by parallel execution of trace samples," in *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, vol. 1, pp. 205–210, Jan 1994.

[82] Lee, J., Jang, H., and Kim, J., "Rpstacks: Fast and accurate processor design space exploration using representative stall-event stacks," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 255–267, Dec 2014.

[83] Leiserson, C. E., "The cilk++ concurrency platform," in *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, (New York, NY, USA), pp. 522–527, ACM, 2009.

[84] Long, D. L. and Clarke, L. A., "Task interaction graphs for concurrency analysis," in *Proceedings of the 11th international conference on Software engineering*, ICSE '89, (New York, NY, USA), pp. 44–52, ACM, 1989.

[85] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K., "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, (New York, NY, USA), pp. 190–200, ACM, 2005.

[86] Lyu, Y.-H., Hong, D.-Y., Wu, T.-Y., Wu, J.-J., Hsu, W.-C., Liu, P., and Yew, P.-C., "Dbill: An efficient and retargetable dynamic binary instrumentation framework using llvm backend," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, (New York, NY, USA), pp. 141–152, ACM, 2014.

[87] Metz, C., "Facebooks new spam-killer hints at the future of coding," September 2015. http://www.wired.com/2015/09/facebooks-new-anti-spam-system-hints-future-coding/.

[88] Microsoft Corporation, *Event Tracing for Windows*, 2015. https://msdn.microsoft.com/en-us/library/windows/desktop/bb968803.aspx.

[89] Microsoft Corporation, *Microsoft Azure Machine Learning*, 2015. https://azure.microsoft.com/en-us/services/machine-learning/.

[90] Molitorisz, K., Karcher, T., Biele, A., and Tichy, W., "Locating parallelization potential in object-oriented data structures," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 1005–1015, May 2014.

[91] Moseley, T., Shye, A., Reddi, V. J., Grunwald, D., and Peri, R., "Shadow profiling: Hiding instrumentation costs with parallelism," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, (Washington, DC, USA), pp. 198–208, IEEE Computer Society, 2007.

[92] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F., "Producing wrong data without doing anything obviously wrong!," in *Proceedings*

of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV, (New York, NY, USA), pp. 265–276, ACM, 2009.

[93] Navada, S., Choudhary, N., Wadhavkar, S., and Rotenberg, E., "A unified view of non-monotonic core selection and application steering in heterogeneous chip multiprocessors," in *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pp. 133–144, Sept 2013.

[94] Nethercote, N. and Seward, J., "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 89–100, ACM, 2007.

[95] Nguyen, D., Lenharth, A., and Pingali, K., "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 456–471, ACM, 2013.

[96] Ofenbeck, G., Steinmann, R., Caparros, V., Spampinato, D., and Puschel, M., "Applying the roofline model," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pp. 76–85, March 2014.

[97] Palframan, D. J., Kim, N. S., and Lipasti, M. H., "Cop: To compress and protect main memory," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 682–693, ACM, 2015.

[98] Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M. A., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., and Sui, X., "The tao of parallelism in algorithms," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 12–25, ACM, 2011.

[99] Poovey, J. A., Railing, B. P., and Conte, T. M., "Parallel pattern detection for architectural improvements," in *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, (Berkeley, CA, USA), pp. 12–12, USENIX Association, 2011.

[100] Railing, B. P., Hein, E. R., and Conte, T. M., "Contech: Efficiently generating dynamic task graphs for arbitrary parallel programs," *ACM Trans. Archit. Code Optim.*, vol. 12, pp. 25:1–25:24, July 2015.

[101] Rane, A. and Browne, J., "Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics," in *Proceedings*

of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, (New York, NY, USA), pp. 147–156, ACM, 2012.

[102] Rico, A., Duran, A., Cabarcas, F., Etsion, Y., Ramirez, A., and Valero, M., "Trace-driven simulation of multithreaded applications," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pp. 87–96, April 2011.

[103] Rico, A., Cabarcas, F., Villavieja, C., Pavlovic, M., Vega, A., Etsion, Y., Ramirez, A., and Valero, M., "On the simulation of large-scale architectures using multiple application abstraction levels," *ACM Trans. Archit. Code Optim.*, vol. 8, pp. 36:1–36:20, Jan. 2012.

[104] Rodrigues, R., Annamalai, A., Koren, I., Kundu, S., and Khan, O., "Performance per watt benefits of dynamic core morphing in asymmetric multicores," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, (Washington, DC, USA), pp. 121–130, IEEE Computer Society, 2011.

[105] Saez, J. C., Fedorova, A., Koufaty, D., and Prieto, M., "Leveraging core specialization via os scheduling to improve performance on asymmetric multicore systems," *ACM Trans. Comput. Syst.*, vol. 30, pp. 6:1–6:38, Apr. 2012.

[106] Samadi, M., Jamshidi, D. A., Lee, J., and Mahlke, S., "Paraprox: Pattern-based approximation for data parallel applications," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 35–50, ACM, 2014.

[107] Sarkar, V. and Simons, B., "Parallel program graphs and their classification," in *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, (London, UK, UK), pp. 633–655, Springer-Verlag, 1994.

[108] Skaletsky, A., Devor, T., Chachmon, N., Cohn, R., Hazelwood, K., Vladimirov, V., and Bach, M., "Dynamic program analysis of microsoft windows applications," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pp. 2–12, March 2010.

[109] Sridharan, S., Gupta, G., and Sohi, G. S., "Adaptive, efficient, parallel execution of parallel programs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), pp. 169–180, ACM, 2014.

[110] Srikanth, S., Railing, B. P., and Conte, T. M., "Dynamically reconfiguring multi-core architectures using task graph based analysis," in *To appear in*

Proceedings of the 24th International Conference on Parallel Architectures and Compilation, PACT '15, (New York, NY, USA), ACM, 2015.

[111] Srinath, S., Ilbeyi, B., Tan, M., Liu, G., Zhang, Z., and Batten, C., "Architectural specialization for inter-iteration loop dependence patterns," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 583–595, Dec 2014.

[112] Srinivasan, S., Zhao, L., Illikkal, R., and Iyer, R., "Efficient interaction between os and architecture in heterogeneous platforms," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 62–72, Feb. 2011.

[113] Srinivasan, S., kurella, N., Koren, I., Rodrigues, R., and Kundu, S., "A runtime support mechanism for fast mode switching of a self-morphing core for power efficiency," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), pp. 491–492, ACM, 2014.

[114] Suleman, M. A., Mutlu, O., Qureshi, M. K., and Patt, Y. N., "Accelerating critical section execution with asymmetric multi-core architectures," *SIGARCH Comput. Archit. News*, vol. 37, pp. 253–264, Mar. 2009.

[115] Tallam, S. and Gupta, R., "Unified control flow and data dependence traces," *ACM Trans. Archit. Code Optim.*, vol. 4, Sept. 2007.

[116] Tian, C., Nagarajan, V., Gupta, R., and Tallam, S., "Dynamic recognition of synchronization operations for improved data race detection," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, (New York, NY, USA), pp. 143–154, ACM, 2008.

[117] Upton, D., Hazelwood, K., Cohn, R., and Lueck, G., "Improving instrumentation speed via buffering," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, (New York, NY, USA), pp. 52–61, ACM, 2009.

[118] Vallejo, E., Beivide, R., Cristal, A., Harris, T., Vallejo, F., Unsal, O., and Valero, M., "Architectural support for fair reader-writer locking," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 275–286, IEEE Computer Society, 2010.

[119] Vandierendonck, H. and De Bosschere, K., "Implicit hints: Embedding hint bits in programs without isa changes," in *Computer Design (ICCD), 2010 IEEE International Conference on*, pp. 364–369, Oct 2010.

[120] Vandierendonck, H., Tzenakis, G., and Nikolopoulos, D. S., "Analysis of dependence tracking algorithms for task dataflow execution," *ACM Trans. Archit. Code Optim.*, vol. 10, pp. 61:1–61:24, Dec. 2013.

[121] Vassenkov, P., "Contech: a shared memory parallel program analysis framework," Master's thesis, Georgia Institute of Technology, Atlanta, Georgia, 2013.

[122] Venkatesh, G., Sampson, J., Goulding-Hotta, N., Venkata, S. K., Taylor, M. B., and Swanson, S., "Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 163–174, ACM, 2011.

[123] Vicik, R., "Private correspondence," Microsoft Corporation, 2015.

[124] Wallace, S. and Hazelwood, K., "Superpin: Parallelizing dynamic instrumentation for real-time performance," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, (Washington, DC, USA), pp. 209–220, IEEE Computer Society, 2007.

[125] Wang, H., Guo, Y., Koren, I., and Krishna, C., "Compiler-based adaptive fetch throttling for energy-efficiency," in *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pp. 112–119, March 2006.

[126] Wang, H., Koren, I., and Krishna, C. M., "An adaptive resource partitioning algorithm for smt processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 230–239, ACM, 2008.

[127] Wang, W., Dey, T., Mars, J., Tang, L., Davidson, J., and Soffa, M., "Performance analysis of thread mappings with a holistic view of the hardware resources," in *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pp. 156–167, April 2012.

[128] Williams, S., Waterman, A., and Patterson, D., "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009.

[129] Wu, J., Tang, Y., Hu, G., Cui, H., and Yang, J., "Sound and precise analysis of parallel programs through schedule specialization," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 205–216, ACM, 2012.

[130] Wu, Q., Martonosi, M., Clark, D. W., Reddi, V. J., Connors, D., Wu, Y., Lee, J., and Brooks, D., "A dynamic compilation framework for controlling microprocessor energy and performance," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, (Washington, DC, USA), pp. 271–282, IEEE Computer Society, 2005.

[131] Yang, X., Blackburn, S. M., and McKinley, K. S., "Computer performance microscopy with shim," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 170–184, ACM, 2015.

[132] Yoo, R. M., Hughes, C. J., Kim, C., Chen, Y.-K., and Kozyrakis, C., "Locality-aware task management for unstructured parallelism: A quantitative limit study," in *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, (New York, NY, USA), pp. 315–325, ACM, 2013.

[133] Zhang, Y., Peng, L., Fu, X., and Hu, Y., "Lighting the dark silicon by exploiting heterogeneity on future processors," in *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, (New York, NY, USA), pp. 82:1–82:7, ACM, 2013.

[134] Zhang, Y., Ozisikyilmaz, B., Memik, G., Kim, J., and Choudhary, A., "Analyzing the impact of on-chip network traffic on program phases for cmps," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 218–226, April 2009.

[135] Zhao, Q., Cutcutache, I., and Wong, W.-F., "Pipa: Pipelined profiling and analysis on multi-core systems," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, (New York, NY, USA), pp. 185–194, ACM, 2008.

[136] Zhao, Q., Cutcutache, I., and Wong, W.-F., "Pipa: Pipelined profiling and analysis on multicore systems," *ACM Trans. Archit. Code Optim.*, vol. 7, pp. 13:1–13:29, Dec. 2010.