

Trace-Based Dynamic Binary Parallelization

A Dissertation

Presented to
the faculty of the School of Engineering and Applied Science
University of Virginia

in partial fulfillment
of the requirements for the degree

Doctor of Philosophy

by


Jing Yang

August

2012

APPROVAL SHEET

The dissertation
is submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy



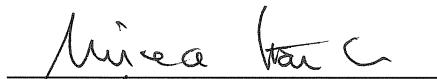
AUTHOR

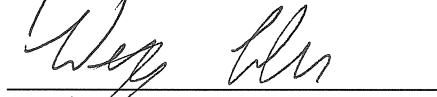
The dissertation has been read and approved by the examining committee:




Advisor



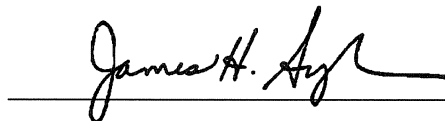






CO-advise.

Accepted for the School of Engineering and Applied Science:



Dean, School of Engineering and Applied Science

August

2012

Abstract

With the number of cores increasing rapidly but the performance per core increasing slowly at best, software must be parallelized in order to improve performance. Manual parallelization is often prohibitively time-consuming and error-prone (especially due to data races and memory-consistency complexities), and some portions of code may simply be too difficult to understand or refactor for parallelization. Most existing automatic parallelization techniques are performed statically at compile time and require source code to be analyzed, leaving a large fraction of software behind. In many cases, some or all of the source code and development tool chain is lost or, in the case of third-party software, was never available. Furthermore, modern applications are assembled and defined at run time, making use of shared libraries, virtual functions, plugins, dynamically-generated code, and other dynamic mechanisms, as well as multiple languages. All these aspects of separate compilation prevent the compiler from obtaining a holistic view of the program, leading to the risk of incompatible parallelization techniques, subtle data races, and resource over-subscription. All the above considerations motivate *dynamic binary parallelization (DBP)*.

This dissertation explores the novel idea of trace-based DBP, which provides a large instruction window without introducing spurious dependencies. We hypothesize that traces provide a generally good trade-off between code visibility and analysis accuracy for a wide variety of applications so as to achieve better parallel performance. Compared to the raw dynamic instruction stream (DIS), traces expose more distant parallelism opportunities because their average length is typically much larger than the size of the hardware instruction window. Compared to the complete control flow graph (CFG), traces only contain control and data dependencies on the execution path which is actually taken. More importantly, while DIS-based DBP typically only exploits fine-grained parallelism and CFG-based DBP typically only exploits coarse-grained parallelism, traces can be used as a unified representation of program execution to seamlessly incorporate the exploitation of both coarse- and fine-grained parallelism.

We develop Tracy, an innovative DBP framework which monitors a program at run time and

dynamically identifies *hot traces*, parallelizes them, and caches them for later use so that the program can run in parallel every time a hot trace repeats. Our experimental results have demonstrated that for floating point benchmarks, Tracy can achieve an average speedup of 2.16x, 1.51x better than the speedup achieved by Core Fusion, one representative of DIS-based DBP techniques. Although the average speedup achieved by Tracy is only 1.04x better than the speedup achieved by CFG-based DBP, Tracy can speed up all floating point benchmarks while CFG-based DBP fails to parallelize three out of eight applications at all. The performance of Tracy is not always better than the performance of existing DIS- and CFG-based DBP techniques. However, it takes the first step to dynamically parallelize the binary executable without using either the raw DIS or the complete CFG. Thus, this dissertation is expected have a broad impact on future researchers who explore other representations of program execution for DBP purposes.

To my parents, Yifeng Yang and Yajuan Dai.

Acknowledgments

For my advisors, Prof. Mary Lou Soffa, Prof. Kamin Whitehouse and Prof. Kevin Skadron, who continuously stimulated my interests in research and encouraged me during those hard times. Their invaluable advice and selfless supports in both professional and personal matters will be forever remembered in the rest of my life.

For my parents, Yifeng Yang and Yajuan Dai, who always respected and supported my choices. Daddy, I believe you are watching over me all the time from heaven.

For my wife, Dr. Meng Wang, who gives me endless love and understands me more than anyone else. I promise that our life will become much better from now on.

For my committee members, Prof. Westley Weimer and Prof. Mircea Stan, who followed my research for several years and always provided insightful suggestions and comments.

For my academic brothers and sisters, Shukang Zhou, Prof. Wei Le, Dr. Apala Guha, Na Zhang, Nguyet Nguyen, Dr. Jason Mars, Dr. Lingjia Tang, Wei Wang, and Tanima Dey, who spent many hours with me in the lab coding, debugging, and fighting for paper deadlines. Your jokes and encouragement made my graduate life much more colorful and enjoyable.

For my friends, Jingbin Zhang, Prof. Shan Lin, Weide Zhang, Dr. Jiakang Lu, Dr. Jiayuan Meng, Dr. Hengchang Liu, Yu Yao, Dr. Tao Long, Ning Zhang, Xiaopu Wang, Dr. Jun Liu, Prof. Roseanne Ford, Dr. Naveen Kumar, Dr. Daniel Williams, Dr. Gogul Balakrishnan, Dr. Franjo Ivancic, Dr. Naoto Maeda, Dr. Aarti Gupta, and Dr. Weihong Li, who spent a lot of good time with me at the University of Virginia and NEC Laboratories America.

Contents

Contents	v
List of Tables	vii
List of Figures	viii
1 Introduction	1
1.1 Problems of State-of-the-Art DBP	2
1.2 Challenges of Trace-Based DBP	4
1.2.1 Trace Construction and Prediction	4
1.2.2 Trace Optimization and Parallelization	5
1.3 Research Overview	6
1.4 Contributions of the Dissertation	9
1.5 Organization of the Dissertation	10
2 Background and Related Work	11
2.1 Evolution of Traces	11
2.2 Software Dynamic Translation	12
2.3 Manual Parallelization	14
2.4 Automatic Parallelization	15
2.4.1 Static Source Parallelization	16
2.4.2 Static Binary Parallelization	17
2.4.3 Dynamic Source Parallelization	17
2.4.4 Dynamic Binary Parallelization	18
3 Limit Study on Parallelizing Traces	21
3.1 Overcoming Inherent Handicaps of Static Parallelization	21
3.2 Limit Study Setup	22
3.2.1 Recording Execution Sequences	23
3.2.2 Analyzing Execution Sequences to Construct Repeating Traces	24
3.2.3 Parallelizing Execution Sequences	26
3.2.4 Modeling Parallel Execution Time	27
3.3 Experimental Results	29
3.3.1 Analysis of Trace Construction	31
3.3.2 Analysis of Trace Parallelization	34
3.3.3 Analysis of Constant Propagation and Value Prediction	34
3.4 Summary	34
4 The Tracy Framework	36
4.1 Motivating Example	37
4.2 Execution Model Justification	38
4.3 Hardware Architecture	40
4.3.1 Supporting Low-Latency Intra-Cluster Communication	41
4.3.2 Supporting Multi-Trace Execution	43
4.4 Summary	44

5	Trace Construction and Prediction	46
5.1	Extending Branch Promotion	46
5.2	Exploiting Hierarchical Code Structures	50
5.2.1	Selecting Starting and Ending Points	52
5.2.2	Appending Retired Instructions	53
5.2.3	Inserting into the Trace Cache	53
5.3	Adaptive Speculation	54
5.4	Experimental Setup	55
5.4.1	Architectures	55
5.4.2	Algorithms	56
5.4.3	Benchmarks	57
5.4.4	Evaluation Methodology	57
5.5	Experimental Results	58
5.6	Summary	62
6	Trace Optimization	63
6.1	Symbolic Execution	65
6.2	Memory Disambiguation	69
6.3	Experimental Setup	71
6.4	Experimental Results	74
6.5	Summary	75
7	Trace Parallelization	76
7.1	Exploiting ILP	76
7.2	Exploiting LLP	77
7.3	Combining ILP and LLP	79
7.4	Experimental Setup	80
7.5	Experimental Results	80
7.5.1	Overall Performance using Different Parallelization Strategies	82
7.5.2	Upgrading to OoO Cores	85
7.5.3	Comparing to DIS- and CFG-Based DBP Techniques	87
7.5.4	Changing System Configurations and Architectural Parameters	90
7.5.5	Isolating Overheads and Benefits	100
7.6	Summary	101
8	Conclusions and Future Work	103
8.1	Merits of the Dissertation	104
8.2	Future Work	105
8.2.1	Balancing and Integrating Coarse- and Fine-Grained Parallelism	106
8.2.2	Resource Allocation	107
8.2.3	Portability and Robustness to Runtime Dynamics	107
8.2.4	Scalability	108
8.2.5	Combining Tracy with Native Parallelization	108
8.2.6	Implementing Tracy on Contemporary Hardware	109
	Bibliography	110

List of Tables

4.1	In the MESI cache coherence protocol, each cache line is in one of four states: 1) modified, 2) exclusive, 3) shared, and 4) invalid.	43
4.2	State transfer of the cache line in the MESI cache coherence protocol when responding to messages initialized from the processor.	43
4.3	State transfer of the cache line in the MESI cache coherence protocol when responding to messages initialized from the snoopy bus.	44
5.1	This table shows 1) the total size of unique traces that commit at least once, 2) the average length of traces that commit in each correct prediction, 3) the percentage of committed traces that are longer than 400 instructions, 4) the trace prediction accuracy, 5) the average size of the candidate trace pool and the average sorted rank of the committed trace in that pool, and 6) the percentage of instructions executed by the unmodified program that are covered by correctly predicted traces.	48
5.2	Parameter definition of the trace construction algorithm.	50
5.3	Architectural parameters of Tracy ₃₄ ⁴ -io2.	56
5.4	Algorithmic parameters of trace construction and prediction.	56
5.5	This table shows trace-related statistical analysis of Tracy ₃₄ ⁴ -io2, including 1) the percentage of instructions executed by the unmodified program that are covered by correctly predicted traces, 2) the number of traces in the trace cache, 3) the trace misprediction rate, 4) the average number of candidate traces for each prediction, and 5) the average length of the trace that commits in each correct prediction.	59
5.6	This table summarizes trace-related statistical analysis of 1) Tracy ₁₀ ⁴ -io2, 2) Tracy ₁₈ ⁴ -io2, and 3) Tracy ₃₄ ⁴ -io2, including 1) the percentage of instructions executed by the unmodified program that are covered by correctly predicted traces, 2) the number of traces in the trace cache, 3) the trace misprediction rate, 4) the average number of candidate traces for each prediction, and 5) the average length of the trace that commits in each correct prediction.	60
7.1	Extra architectural parameters of 2-issue OoO cores.	80
7.2	This table shows the percentage of instructions executed by the unmodified program that are covered by correctly predicted traces in which LLP is exploited over instructions that are covered by any correctly predicted traces.	82
7.3	This table summarizes the performance of 1) Tracy ₃₄ ⁴ -io2, 2) Tracy ₃₄ ⁴ -ooo2, and 3) Tracy ₃₄ ⁴ -ooo4. Results are normalized to ST-io2, ST-ooo2, and ST-ooo4, respectively. Two metrics are evaluated for each category of benchmarks: 1) the number of programs with improved performance, and 2) the speedup averaged over programs with improved performance.	85

List of Figures

1.1	Design spectrum of the representation of program execution on which parallelization is performed. Compared to DIS and complete CFG, Trace has the potential to provide a large instruction window without introducing spurious dependencies.	3
2.1	Generic SDT systems contain three algorithmic components and transparently manipulate the binary executable while it is running.	13
2.2	Taxonomy of automatic parallelization techniques based on their applicability. The code that cannot be parallelized is listed under each category.	15
3.1	Analysis of the trace (dashed arrow) produces fewer true dependencies than analysis of the CFG, leading to improved parallel performance.	22
3.2	The idealized trace construction algorithm finds the most frequently repeating patterns of instructions in the entire execution sequence, as shown in the example. The handicapped version does not construct traces across boundaries between application and library code.	25
3.3	The standard T-DBP achieves an average speedup of 9.36x and 22.34x over sequential execution for integer and floating point benchmarks, respectively. When all handicaps are artificially emulated, the average speedup shrinks to 4.68x for integer benchmarks and 9.36x for floating point benchmarks. Traditional constant propagation and perfect value prediction could potentially improve execution speed by another factor of 1.97x for integer benchmarks and 3.49x for floating point benchmarks on average.	30
3.4	An average of 10.32% basic blocks or 8.40% instructions for integer benchmarks and 17.91% basic blocks or 11.12% instructions for floating point benchmarks belong to libraries.	31
3.5	The average trace length is 235 basic blocks for integer benchmarks and 4,565 basic blocks for floating point benchmarks.	32
3.6	An average of 0.02% basic blocks for integer benchmarks and 0.19% basic blocks for floating point benchmarks are not formed into traces.	33
4.1	Tracy takes a sequential instruction stream and transparently converts it to a set of parallel instruction streams.	37
4.2	Tracy uses one core for trace management plus sequential execution, and the remaining cores for speculative execution of parallelized candidate traces.	38
4.3	Based on multi-trace execution, Tracy can successfully parallelize some applications when CFG-based DBP techniques fail to.	39
4.4	Tracy assumes that a many-core chip is organized into master clusters with a specially-instrumented core and slave clusters that contain synchronization arrays.	41
4.5	Each entry in the synchronization array is correlated to a unique register or memory reference that needs to be synchronized. While the actual value of each synchronized register is explicitly transferred through the array, a boolean value is just enough for a pair of dependent memory references to maintain the correct order.	42

5.1	Tracy constructs traces at the retire stage of each instruction. It stores traces in main memory and their metadata in the set associative trace cache on chip.	51
5.2	Tracy starts to exploit code structures in the outermost scope, and only enters the next level if necessary.	52
6.1	Tracy optimizes and parallelizes the constructed trace in five steps.	64
6.2	Tracy divides all MIPS instructions into four categories: 1) memory loads, 2) memory stores, 3) instructions that only perform integer arithmetic and logical operations (including control transfer instructions that test integer registers), and 4) all other instructions. Tracy uses different strategies to symbolically evaluate instructions in different categories.	67
6.3	Tracy performs symbolic evaluation and memory disambiguation before parallelization to eliminate unnecessary data dependencies (both register and memory) or increase data dependency lengths, exposing more parallelism opportunities.	68
6.4	Tracy inserts extra dependencies to ensure that every monitored base register receives the correct value.	70
6.5	This figure shows the speedup of Tracy ₃₄ ⁴ -io2 when 1) optimization is disabled, and 2) optimization is enabled. Results are normalized to ST-io2.	72
6.6	This figure shows the optimization-only speedup of Tracy ₃₄ ⁴ -io2 when 1) memory disambiguation is disabled, and 2) memory disambiguation is enabled. Results are normalized to ST-io2.	73
7.1	Accumulator expansion replaces the single shared accumulator with multiple private accumulators.	78
7.2	Dependent code motion pushes every producer to be executed earlier and every consumer to be executed later.	78
7.3	This figure shows the speedup of Tracy ₃₄ ⁴ -io2 when it is configured to exploit 1) LLP only, 2) ILP only, and 3) both LLP and ILP. Results are normalized to ST-io2.	81
7.4	This figure shows the energy consumption of Tracy ₃₄ ⁴ -io2 when it is configured to exploit both LLP and ILP. Results are normalized to ST-io2. The adjusted energy consumption is achieved by counting in “system leakage” from other machine components and power supply inefficiencies.	83
7.5	This figure shows the speedup of Tracy ₃₄ ⁴ -ooo2 when it is configured to exploit both LLP and ILP. Results are normalized to ST-ooo2.	86
7.6	This figure shows the speedup of Tracy ₃₄ ⁴ -ooo4 when it is configured to exploit both LLP and ILP. Results are normalized to ST-ooo4.	86
7.7	This figure compares the performance of Core Fusion, one representative of DIS-based DBP techniques, with Tracy ₃₄ ⁴ -ooo2 and Tracy ₆₆ ⁸ -ooo2 when they are configured to exploit both LLP and ILP. Results are normalized to ST-ooo2.	88
7.8	This figure compares the performance of CFG-based DBP with Tracy ₁₈ ² -io2 when it is configured to exploit both LLP and ILP. Results are normalized to ST-io2.	89
7.9	This figure shows the speedup of 1) Tracy ₁₀ ⁴ -io2, 2) Tracy ₁₈ ⁴ -io2, and 3) Tracy ₃₄ ⁴ -io2 when they are configured to exploit both LLP and ILP. Results are normalized to ST-io2.	91
7.10	This figure shows the speedup of 1) Tracy ₁₈ ² -io2, 2) Tracy ₃₄ ⁴ -io2, and 3) Tracy ₆₆ ⁸ -io2 when they are configured to exploit both LLP and ILP. Results are normalized to ST-io2.	92
7.11	This figure shows the speedup of Tracy ₃₄ ⁴ -io2 when the synchronization array has access latency of 1) 8 clock cycles, 2) 4 clock cycles, and 3) 1 clock cycle. The system is configured to exploit both LLP and ILP. Results are normalized to ST-io2.	93
7.12	This figure shows the speedup of Tracy ₃₄ ⁴ -io2 when the backbone bus has transfer delay of 1) 12 clock cycles, 2) 6 clock cycles, and 3) 3 clock cycles. The system is configured to exploit both LLP and ILP. Results are normalized to ST-io2.	94

7.13	This figure shows the speedup of Tracy ₃₄ ⁴ -io2 when the spill space has access latency of 1) 3 clock cycles, 2) 2 clock cycles, and 3) 1 clock cycle. The system is configured to exploit both LLP and ILP. Results are normalized to ST-io2.	95
7.14	This figure shows the speedup of Tracy ₃₄ ⁴ -io2 when 1) symbolic evaluation is disabled, and 2) symbolic evaluation is enabled. The system is configured to exploit both LLP and ILP. Results are normalized to ST-io2.	97
7.15	This figure shows the speedup of Tracy ₃₄ ⁴ -io2 when 1) memory disambiguation is disabled, and 2) memory disambiguation is enabled. The system is configured to exploit both LLP and ILP. Results are normalized to ST-io2.	98
7.16	This figure shows the execution time overhead of Tracy ₃₄ ⁴ -io2 caused by mis-speculation and program state transfer. The system is configured to exploit both LLP and ILP. Useful execution includes the time spent on both sequential execution and parallel execution that is speculated correctly.	99

Chapter 1

Introduction

As a consequence of the diminishing returns for increasing complexity, microarchitecture designers have started to increase the number of cores on a single chip instead of trying to increase its single-threaded performance. Computers with four to eight cores are already ubiquitous and trends suggest that core counts will continue to grow for the foreseeable future [1]. While the computational capability of the chip continues to double every 18 months in accordance with Moore's Law, the performance of individual cores has largely stagnated due to limitations on area, power consumption, and heat dissipation.

With the number of cores increasing rapidly but the performance per core increasing slowly at best, software must be parallelized in order to improve performance. Manual parallelization typically yields the best speedups because the programmer can choose new algorithms and data structures that are more amenable to parallelism. However, manual parallelization is often prohibitively time-consuming and error-prone (especially due to data races and memory-consistency complexities), and some portions of code may simply be too difficult to understand or refactor for parallelization. Code is only parallelized when the return on investment is sufficient.

There has also been considerable research on automatic parallelization. However, most existing automatic parallelization techniques are performed statically (i.e., at compile time) and require source code to be analyzed, suffering three serious problems. First, in many cases, some or all of the source code and development tool chain is lost or, in the case of third-party software, was never available. During the Y2K crisis, it was estimated that some companies were missing as much as 60 percent of their source code [2]. Second, modern applications are assembled and defined at run time, making use of shared libraries, virtual functions, plugins, dynamically-generated code, and other dy-

dynamic mechanisms, as well as multiple languages. All these aspects of separate compilation prevent the compiler from obtaining a holistic view of the program, leading to the risk of incompatible parallelization techniques, subtle data races, and resource over-subscription. Third, compile-time analysis has to conservatively respect all control and data dependencies on the control flow graph (CFG). This deters parallelization, because many of these dependencies may not be on the execution path which is actually taken. All the above considerations motivate binary code parallelization at run time, which we call *dynamic binary parallelization (DBP)*. Without effective techniques that can operate on binary code, a large fraction of software will be left behind. And without the ability to parallelize at run time, opportunities for parallelism are curtailed.

1.1 Problems of State-of-the-Art DBP

Prior research on DBP has been largely limited. Existing DBP technologies are generally divided into two main categories: parallelizing the raw dynamic instruction stream (DIS) [3, 4, 5] and parallelizing the dynamically-generated CFG [6, 7, 8, 9, 10].

DIS-based techniques use extra hardware to combine multiple cores to work cooperatively as a wider core. Native out-of-order (OoO) execution could also be considered as a DIS-based technique, which has been widely adopted by modern microarchitectures. Focusing on exploiting instruction level parallelism (ILP), this technology has wide applicability, because ILP typically exists throughout the entire program (with different amounts). Limited by branch prediction accuracy and instruction window size, however, this technology generally fails to exploit distant or coarse-grained parallelism, resulting in relatively mediocre speedups.

On the other hand, CFG-based techniques expose a global view of the program and allow discovery of more coarse-grained loop and thread level parallelism (LLP and TLP), which have the potential to produce much larger speedups. However, analysis on the CFG must be conservative and consider the large number of possible paths of program execution, some of which may be rarely executed. This requires the compiler to respect control and data dependencies that do not appear in the actual execution path, inhibiting parallelization and requiring extensive speculation. When source code is not available, this problem is exaggerated due to the lack of high-level information (e.g., types, variables, data structures), which is essential to achieve accurate alias analysis. Furthermore, when coarse-grained parallelism is hard to exploit, CFG-based techniques typically lack the capability to extract fine-grained parallelism instead and just execute the program sequentially. Thus, it is

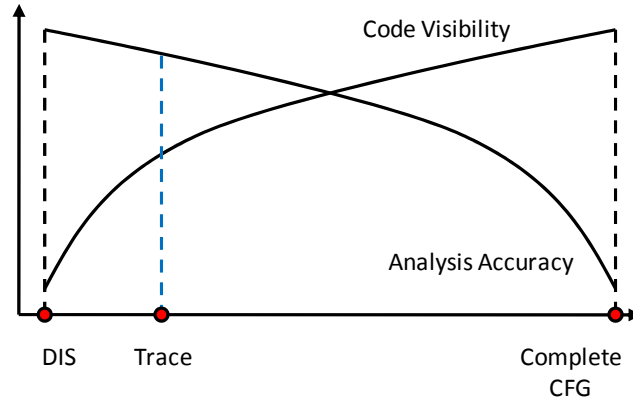


Figure 1.1: Design spectrum of the representation of program execution on which parallelization is performed. Compared to DIS and complete CFG, Trace has the potential to provide a large instruction window without introducing spurious dependencies.

not surprising that most of the existing CFG-based techniques [6, 7, 10] have failed to parallelize at least half applications in the selected benchmark suite.

Although DIS- and CFG-based techniques are complementary to each other, no prior research has tried to implement both technologies under a unified system. Such a system may achieve large speedups from code regions that contain coarse-grained LLP or TLP, and exploit ILP from the remainder of the program. This combination is quite vital. As Amdahl’s Law shows, even a small fraction of non-parallelizable code can drastically inhibit overall speedups. Asymmetric architectures [11, 12] try to address this problem by providing one or more large, OoO cores on the chip for sequential execution modes, and a larger number of simple, in-order (IO) cores to maximize throughput. Due to chip area constraints, however, this organization reduces the number of total cores on the chip, hurting the performance of code regions where plentiful LLP can be exploited. The OoO core may also not be available when needed, if multiple applications are sharing the CPU. Furthermore, when LLP or TLP is limited, there will often be more coarse-grained tasks needing further acceleration via ILP than there are OoO cores. Finally, prior research [3, 4, 5] suggests that cooperative work of multiple IO cores can generally outperform or at least compete with an OoO core, which may obviate the need for an asymmetric architecture in the first place.

Does any representation of program execution provide a large instruction window without introducing spurious dependencies? Figure 1.1 illustrates the design spectrum of the representation of program execution on which parallelization is performed. Theoretically, any point on the design spectrum between the two extremes (DIS and complete CFG) may be the one that achieves the optimal trade-off between code visibility and analysis accuracy. Unlike compile-time parallelization, DBP has the potential to construct *dynamic* CFGs from the instruction stream by only considering

execution paths which are actually taken in each particular run. These *partial* CFGs typically represent the execution of smaller code structures (e.g., part of a function instead of the entire function) than *static* CFGs in order to maintain a practical number of unmerged control flows for aggressive parallelization. As the design point shifts to the right on the design spectrum, each dynamic CFG represents the execution of larger code structures and merges more execution paths until the complete whole-program CFG is constructed.

In practice, many programs tend to frequently repeat long sequences of instructions called *hot traces*, which have the potential to provide a generally good trade-off between code visibility and analysis accuracy for a wide variety of applications so as to achieve better parallel performance. First, traces only represent the execution path which is actually taken, eliminating spurious control and data dependencies to the highest possible extent. Second, traces can act as a unified representation of program execution to seamlessly incorporate the exploitation of both coarse- and fine-grained parallelism. For traces that comprise multiple loop iterations, LLP can be exploited with higher priority. By only considering a single execution possibility, many loop-carried dependencies are simply eliminated and more accurate alias analysis can be achieved. As a result, more code regions may contain exploitable LLP, greatly increasing the applicability of existing CFG-based techniques. For the remainder of the program that is less parallelizable, long traces may still expose distant ILP opportunities. The average length of traces is typically much larger than the size of the hardware instruction window used in existing DIS-based techniques.

1.2 Challenges of Trace-Based DBP

The major challenge of trace-based DBP is constructing high-quality traces that provide a large instruction window without introducing spurious dependencies. Due to Amdahl's Law, these traces should also cover a large portion of dynamic instructions in order to produce large overall speedups. Another challenge is customizing algorithms that are most suitable to optimize and parallelize binary code in trace format. The following sections will describe these challenges in detail.

1.2.1 Trace Construction and Prediction

Prior research [13] has demonstrated that the return of dynamic binary optimization (DBO) diminishes when traces are longer than 200 basic blocks. For parallelization purposes, however, traces have to be *as long as possible* to expose more distant parallelism opportunities. Furthermore, traces have to be *logically atomic*. They should have a single entry point, a single exit point, and the control

flow cannot exit prematurely through so-called *side exits*. Thus, analysis can ignore all unnecessary control and data dependencies, enabling more aggressive parallelization. This atomicity property necessitates speculative execution to recover program state when a trace deviates from the execution path which is actually taken. This is usually easier and less costly than more fine-grained recovery code required to support side exits. Finally, traces of different program phases should all meet the above two requirements so that the program can run in parallel most of the time.

The dilemma, however, is that the longer a trace is, the more difficult to achieve high speculation accuracy. To the best of our knowledge, most existing technologies do not support the atomicity property of traces [14, 15, 16, 17, 18, 19, 20]. rePlay [21] does perform DBO on short atomic traces (16 to 256 instructions long), but they are not suitable for parallelization purposes. Before the many-core era, some systems were proposed [22, 23, 24, 25] to use hardware-only technologies to speculate multiple consecutive atomic traces and execute them simultaneously on different functional units. In order to achieve reasonable speculation accuracy, however, these systems construct very short traces, which necessitates ultra-low communication latency to support program state transfer. Furthermore, these systems rely on fine-grained selective recovery from frequent mis-speculation. While suitable for simultaneous-multithreaded or clustered microarchitectures, neither requirement can be easily satisfied on many-core architectures.

1.2.2 Trace Optimization and Parallelization

Accurate alias analysis is usually the key factor to enable effective program optimization and parallelization. Lacking high-level information (e.g., types, variables, data structures), it is extremely difficult to disambiguate memory references when source code is not available. Thus, most existing software dynamic translation (SDT) systems, such as Dynamo [14], DynamoRio [15], Transmeta [26], and Daisy [27], only perform alias analysis in the form of *instruction inspection*, which disambiguates two memory references if they access either different memory regions or their addresses have the same base register and different offsets. As has been demonstrated by prior research [28], instruction inspection can only disambiguate one-third of all memory references in SPEC CUP2000 integer benchmarks, greatly restricting aggressive code transformations.

Compilers typically rely on various data-flow analyses [29] to optimize programs. They set up data-flow equations for each node of the CFG and solve them by repeatedly calculating the output from the input locally at each node until the entire system stabilizes, i.e., it reaches a fixpoint. In order to handle the large number of execution paths represented by the CFG, diverged program states

are conservatively merged at certain joint points. Furthermore, the name space of data-flow functions is typically based on lexical names of variables, leaving many optimization opportunities behind. Some frameworks [30, 31] have been developed to achieve scalable path-based value-flow analysis, but all of them are targeted to bug detection instead of code transformations. On the contrary, an atomic trace not only represents a single execution path, but also has no side exits so that all control dependencies and derived data dependencies among its instructions can be ignored. Thus, it is both important and necessary to design heavyweight but powerful optimization algorithms to fully exploit the atomicity property of traces, which has never been studied by existing optimization systems that also leverage atomic traces [21, 32].

Due to limited trace length, existing trace-based parallelization systems [33, 34] only focus on exploiting local ILP. However, prior research [35] has demonstrated that there is no dominant type of parallelism. The contribution of each type of parallelism varies widely across applications. Thus, one major prerequisite to achieve the greatest speedups is to accurately identify the most appropriate type of parallelism that should be exploited in each code region. Parallelization at the trace level further increases the difficulty of exploiting hybrid parallelism. For the same code region that has complicated control flows, different traces may have quite different characteristics, which need customized parallelization algorithms to meet the specific requirements.

1.3 Research Overview

This dissertation explores the novel idea of trace-based DBP, which provides a large instruction window without introducing spurious dependencies. We hypothesize that traces provide a generally good trade-off between code visibility and analysis accuracy for a wide variety of applications so as to achieve better parallel performance. Compared to DIS-based DBP, trace-based DBP can exploit more distant parallelism because the average length of traces is typically much larger than the size of the hardware instruction window. Compared to CFG-based DBP, trace-based DBP does not need to respect control and data dependencies that are not on the execution path which is actually taken. More importantly, while DIS-based DBP typically only exploits fine-grained parallelism and CFG-based DBP typically only exploits coarse-grained parallelism, traces can be used as a unified representation of program execution to seamlessly incorporate the exploitation of both coarse- and fine-grained parallelism.

Before developing any specific design of trace-based DBP, we first conduct a limit study to: 1) identify the performance limits of trace-based DBP, and 2) explain *why* trace-based DBP performs

as it does. The first goal is to set up the performance upper bound so that any following specific design can be judged to determine whether it has fully exploited the benefits of trace-based DBP. The second goal is to identify the unique and powerful characteristics of trace-based DBP that enable it to achieve substantial speedups. We highlight these characteristics by comparing trace-based DBP to static parallelization which is typically performed by the compiler.

We analyze the performance limits of trace-based DBP by making three idealizations about the hardware and algorithms: 1) the program runs on a many-core architecture with an unbounded number of cores and an unlimited, shared L1 cache, 2) the trace construction algorithm can always identify the most frequently repeating patterns of instructions that will occur in a particular run of the program, and 3) when the program reaches a repeating trace, the trace prediction algorithm can always correctly predict the trace that is about to run.

The limit study performs a four step process for each benchmark: 1) record the complete execution sequence of the program, 2) analyze the recording offline to identify the frequently repeating traces, 3) create a new execution sequence by replacing each trace in the original execution sequence with the parallelized version, and 4) analyze the parallel execution time of the new execution sequence using a model of a shared-memory many-core architecture.

We then develop Tracy, an innovative DBP framework which monitors a program at run time and dynamically identifies hot traces, parallelizes them, and caches them for later use so that the program can run in parallel every time a hot trace repeats. In order to achieve the greatest speedups over sequential execution, Tracy has to construct high quality traces as well as to customize the most suitable algorithms to optimize and parallelize these traces.

High quality traces have to simultaneously satisfy four requirements, which can be contrary to one another. First, traces have to be as long as possible to expose more distant parallelism opportunities. Second, traces have to be logically atomic. They should have a single entry point, a single exit point, and the control flow cannot exit prematurely through side exits. Thus, analysis can ignore all unnecessary control and data dependencies, enabling more aggressive parallelization. Third, traces have to be predicted accurately so that valuable CPU cycles and energy are not wasted on executing incorrect execution paths. Fourth, traces have to cover a large portion of dynamic instructions so as to produce large overall speedups.

Based on the above insights, we exploit the unique power of many-core architectures by launching multiple traces and executing them simultaneously on idle cores. The major insight is that in many cases, speculation accuracy can be *dramatically* increased by only trying a *very small* number of candidate traces. We also develop an innovative trace construction algorithm that holistically

balances among trace length, speculation accuracy, and coverage of dynamic instructions. Tracy constructs the longest traces that can be accurately speculated on the available number of cores. In certain code regions that have complicated control flows, Tracy stops constructing traces and executes these code regions sequentially. More specifically, we leverage the hierarchical code structures (e.g., functions, loops, basic blocks) to define the starting and ending points of each trace. In order to maximize their length, traces are initially restricted to start and end at the outermost functions or loops. During program execution, if a code structure shows unpredictable internal execution paths, it is abandoned and the next level of inner code structures is used instead. If the innermost code structure still has complicated control flows that are hard to predict, the corresponding code regions are executed sequentially.

We develop two major optimizations to optimize traces that have been constructed, namely *symbolic evaluation* and *memory disambiguation*. The functionality of these optimizations is not only to directly produce speedups, but more importantly, to prepare the code for future parallelization. Thus, the performed code transformations may be suboptimal for increasing the program performance by themselves, but they reformat the code to be more amenable to parallelism. Furthermore, these optimizations are designed to fully exploit the atomicity property of traces, within the confines of the underlying architectural support.

Symbolic evaluation [31] assigns a symbolic value to each defined register or memory location. Code analysis and transformations are then performed through symbolically executing each program path and updating these values. The path sensitivity characteristic of symbolic evaluation is not scalable, and thus has not been used by the compiler that needs to optimize the entire CFG. However, because symbolic evaluation performs path-sensitive program analysis and data-flow information is based on symbolic values instead of lexical names, it performs more precise program analysis than traditional data-flow analysis [29]. For memory disambiguation, we follow the idea of [28] and divide all memory references into groups with different base registers. If the ranges of addresses covered by two groups are *disjoint*, memory references in one group are guaranteed not to alias with those in the other group. Disambiguating memory references at the group granularity is necessary because the sequential order of all memory references is lost after parallelization, which, however, may introduce false positives due to the approximation of memory addresses.

Our goal is not to develop totally innovative parallelization algorithms, but to customize off-the-shelf algorithms to make them suitable for parallelizing atomic traces. For exploiting ILP, Tracy adopts the traditional list scheduling algorithm [36] to partition and schedule instructions among different cores. Unlike the original algorithm that only reorders instructions on the same core, we

have to minimize inter-core synchronization overhead as well as cache coherence traffic on the many-core architecture. For exploiting LLP, Tracy performs two major code transformations, *accumulator expansion* and *dependent code motion*, to eliminate loop-carried dependencies or at least to increase the execution overlap of multiple threads so as to achieve better parallel performance.

We also leverage traces as the unified representation of program execution to exploit both coarse- and fine-grained parallelism. This combination is quite vital. As Amdahl's Law shows, even a small fraction of non-parallelizable code can drastically inhibit overall speedups. When both types of parallelism are available, Tracy selects the optimal strategy at the trace level. It first parallelizes the trace by exploiting LLP, which has the potential to produce larger speedups. If limited LLP exists, however, Tracy extracts ILP from the trace instead.

1.4 Contributions of the Dissertation

The contributions of the dissertation are listed as follows:

- A limit study that not only identifies the performance limits of trace-based DBP, but also explains *why* trace-based DBP performs as it does. It sets up the performance upper bound so that any following specific design can be judged to determine whether it has fully exploited the benefits of trace-based DBP. It also identifies the unique and powerful characteristics of trace-based DBP that enable it to achieve substantial speedups.
- A capture-analysis framework that efficiently records program execution and provides a user-friendly environment to effectively analyze the recording. This framework can be widely applicable to future studies on dynamic trace-based systems.
- An innovative trace-based DBP framework named Tracy which provides a large instruction window without introducing spurious dependencies. Compared to DIS-based DBP, it can exploit more distant parallelism and compared to CFG-based DBP, it only needs to respect control and data dependencies on the execution path which is actually taken.
- A general trace construction algorithm that holistically balances among trace length, speculation accuracy, and coverage of dynamic instructions. This algorithm constructs the longest traces that can be accurately speculated on the available number of cores and can be readily adopted by other dynamic trace-based systems.

- Two code optimizations, symbolic evaluation and memory disambiguation, that are specifically designed for atomic traces. They not only directly produce speedups, but also transform the code to be more amenable to parallelism, which is usually more important.
- A unified parallelization system that uses customized algorithms to extract both coarse- and fine-grained parallelism from atomic traces and selects the optimal strategy based on the estimated parallel performance.

1.5 Organization of the Dissertation

The remainder of the dissertation is organized as follows. Chapter 2 provides background on the evolution of traces and different parallelization technologies to facilitate understanding of the subsequent chapters. Chapter 3 discusses a limit study to prove the feasibility of trace-based DBP and Chapter 4 then presents Tracy, an innovative trace-based DBP framework which leverages multi-trace execution to exploit the unique power of many-core architectures. After that, Chapters 5 to 7 describe trace construction and prediction, trace optimization, and trace parallelization, respectively, which are the three most important functionalities of Tracy. Finally, Chapter 8 concludes the dissertation and discusses future work.

Chapter 2

Background and Related Work

Two key concepts of this dissertation are *trace* and *parallelization*. Thus, we organize the background chapter based on these two concepts. Under the trace concept, we first describe its evolution during the last three decades and then discuss how traces are typically constructed during run time using SDT systems. Under the parallelization concept, we discuss both manual parallelization and automatic parallelization. Automatic parallelization is further divided into four categories, each of which is described in detail.

2.1 Evolution of Traces

Traces have long been used to improve program performance. While VLIW and superscalar processors need sufficient ILP to fully exploit the processing power of the underlying parallel architecture, ILP within basic blocks is limited for programs that have complicated control flows. Thus, optimizations across basic block boundaries are needed. Based on profiling information, the initial traces are constructed by the compiler, which removes constraints due to unimportant execution paths and links basic blocks together following the most frequently executing path. These traces contain both side entrances and side exits, where the control flow can enter and exit the trace arbitrarily. For several different architectures, trace scheduling [37, 38] has been proposed to exploit more ILP by performing code motion on these long sequences of instructions. However, the existing side entrances require very complex bookkeeping information to schedule instructions across basic block boundaries, if at all possible.

In order to remove the problems of side entrances, tail duplication has been proposed to ensure that the control flow can only enter from the top of the trace. More specifically, the tail portion of

the trace is duplicated from the first side entrance to the end, with all side entrances being moved to the corresponding duplicated basic blocks. Such reformatted traces with a single entry and multiple exits are called superblocks [16, 39]. Superblocks are not only constructed by the compiler based on profiling information, but are also leveraged by a variety of modern microarchitectures and SDT systems at run time. For example, the trace cache [40] increases instruction fetch width by caching dynamic instruction sequences; the trace processor [25] speeds up control prediction by speculating on traces instead of branches; DBO systems [14, 15, 18, 20, 21] exploit optimization opportunities on traces which are not available by statically analyzing the CFG.

Although superblocks do not have side entrances, the side exits still prevent instructions from being freely scheduled across basic block boundaries. For example, an instruction cannot be moved above a preceding side exit if it is used before it is redefined when the side exit is taken. Thus, prior research [21, 32] has introduced atomic traces, which have a single entry point and a single exit point. Furthermore, these reformatted traces encapsulate only a single flow of control. If any instruction within the trace can not execute, all executed instructions within the trace are squashed. This atomicity property provides more flexibility for performing beneficial code transformations. Instructions within the trace are not control dependent on one another and can be scheduled across any basic block boundaries as long as data dependencies are respected.

Another enhancement of superblocks is called hyperblocks [17, 19], which represent multiple execution paths simultaneously. Instructions from different execution paths are guarded by hardware-supported predicates to maintain the correct control flow at run time. The motivation behind hyperblocks is to group many basic blocks from different execution paths into a single manageable code region for compiler optimization. Thus, for programs without heavily biased branches, hyperblocks have wider applicability than superblocks to enable beneficial compile-time code transformations that do not exist on the original CFG. However, hyperblocks also contain side exits and thus do not have the atomicity property.

2.2 Software Dynamic Translation

As described above, traces are initially constructed by the compiler based on profiling information. Recently, however, SDT systems become increasingly popular to extract traces directly from the instruction stream while the program is running. Figure 2.1 illustrates how an SDT system is positioned under the application to intercept the native instruction stream at run time. Three algorithmic components form the foundation of a generic SDT implementation: 1) the *translator*

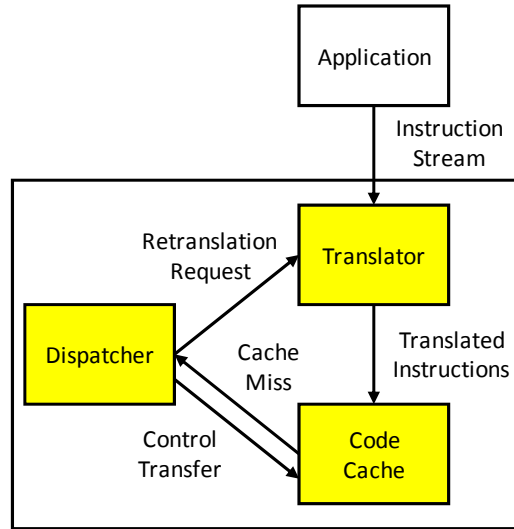


Figure 2.1: Generic SDT systems contain three algorithmic components and transparently manipulate the binary executable while it is running.

translates instructions from the binary executable into traces that will run on the actual hardware, often interjecting or altering instructions en route, 2) the *code cache* holds these commonly executed traces so that they can be executed again without re-translation, and 3) the *dispatcher* redirects the control flow to either the next trace, if it is already in the trace cache, or to the translator which re-translates the target address. In the mean time, the original program either continues running without being translated or is suspended until the new trace is completed.

SDT systems have been implemented purely in software and executed on commodity microarchitectures for diverse purposes, such as optimization [14, 15], ISA translation [27], profiling [41, 42], or security monitoring [43, 44]. Due to the large overhead of software-based dynamic instrumentation [41, 42], however, most systems [27, 41, 42, 43, 44] do not profile the running program and terminate the trace at any conditional branch or jump that has multiple targets. Other systems [14, 15] use the next executed tail (NET) algorithm to construct superblocks, which only needs very lightweight instrumentation. The NET algorithm has two phases, the profiling phase and the collection phase [45]. In the profiling phase, each conditional branch that is a backedge is instrumented. A counter is maintained for each of these backedges and incremented every time that backedge is taken. When this counter reaches a predetermined threshold, the next executed superblock is collected. In the collection phase, the code is instrumented and monitored basic block by basic block until another backedge is encountered. The NET algorithm has two significant shortcomings: 1) it is possible to collect a cold path during the collection phase even if a hot path was primarily responsible for reaching the backedge during the profiling phase, and 2) the constructed

hyperblocks cannot span any function invocations/returns and loops, and thus are typically only two to three basic blocks long. Prior research [45] has further demonstrated that for the SPEC CPU2006 benchmark suite, the superblocks constructed by the NET algorithm only account for less than 40% of dynamic instructions and for more than 80% of the times, the control flow leaves the running superblock prematurely via its side exits.

In order to improve trace quality with low runtime overhead, prior research [25, 40] has proposed to implement dynamic instrumentation in hardware by directly integrating trace construction with non-critical stages of instruction pipeline execution. This technology has been adopted by a variety of DBO systems [18, 20, 21], for which increasing program speed is the top priority. For example, both [18] and [20] profile the behavior (i.e., execution frequency and target bias) of every conditional branch, which would cause prohibitive runtime overhead if using software-based dynamic instrumentation. Highly biased hot branches are then identified and analyzed to generate those dominant execution paths. As has been reported by [18], 12% of the constructed hyperblocks are more than 50 instructions long and 89% of them cover at least 100 million dynamic instructions during program execution. rePlay [21, 46] takes a further step by associating each conditional branch with the global branch history [47]. This context sensitive information separates each conditional branch into instances based on the execution path leading up to itself. Once separated this way, a greater number of conditional branches tend to exhibit biased behavior. For integer applications in the SPEC CPU2000 benchmark suite, rePlay is capable of constructing atomic traces of 102 instructions on average, which results in optimization effectiveness [13].

2.3 Manual Parallelization

With the number of cores increasing rapidly but the performance per core increasing slowly at best, software must be parallelized in order to improve performance. A variety of parallel programming frameworks such as OpenMP [48], Chapel [49], and Axum [50] have been proposed to help software engineers fully exploit this increased processing power by executing threads on multiple cores simultaneously. Based on these frameworks, specific parallel libraries [51, 52] have also been proposed to facilitate efficient software development in different domains. Such manual parallelization typically yields the best speedups because the programmer can choose new algorithms and data structures that are more amenable to parallelism.

For a variety of reasons, however, many programs will not be completely parallelized and will continue to have both parallel and sequential modes of execution. Software engineers may not have

	Static	Dynamic
Source	<ul style="list-style-type: none"> ➤ Legacy Software ➤ Third-Party Software ➤ Multi-Language Software ➤ Dynamic Linking and Loading ➤ Self-Modifying Code 	<ul style="list-style-type: none"> ➤ Legacy Software ➤ Third-Party Software ➤ Multi-Language Software
Binary	<ul style="list-style-type: none"> ➤ Dynamic Linking and Loading ➤ Self-Modifying Code 	If the software can execute, then we can parallelize it !

Figure 2.2: Taxonomy of automatic parallelization techniques based on their applicability. The code that cannot be parallelized is listed under each category.

the source code for some or all parts of the program because it was lost or because it uses third-party libraries. Furthermore, manual parallelization is often prohibitively time-consuming and error-prone, especially due to data races and memory-consistency complexities. It has been estimated that the efforts to analyze, fix, and test existing software due to the Y2K bug alone have cost about \$2 billion in the 1990s [53], and rewriting code to find opportunities for parallelism would be a much larger and more challenging task. Finally, some portions of code may be too difficult to understand or refactor for parallelization and other portions of code would simply not yield enough speedups to justify manual parallelization.

2.4 Automatic Parallelization

In order to extricate software engineers from manual parallelization, there has been considerable research on automatic parallelization. We classify automatic parallelization techniques into four different categories based on two orthogonal criteria: 1) whether the technique is performed statically at compile time or dynamically at run time, and 2) whether the technique analyzes source code or binary code. Figure 2.2 shows the code that cannot be parallelized by each category of techniques. Source code parallelizers cannot handle legacy software, third-party software, and software written in different languages. On the other hand, static parallelizers cannot handle dynamic linking/loading and self-modifying code. Thus, only DBP techniques can parallelize any code. If the software can

execute, then they can parallelize it. In the following sections, we will describe the four categories of automatic parallelization techniques in detail.

2.4.1 Static Source Parallelization

Static parallelization techniques typically analyze source code to extract parallelism at compile time [33, 34, 35, 54, 55, 56, 57, 58, 59, 60]. These techniques have been used to exploit ILP [33, 34], LLP [54, 56], TLP [55, 57, 58, 59, 60], or even a combination of them [35]. Furthermore, some techniques are designed to parallelize general-purpose software on commodity machines [55, 56], while others are specially customized for special microarchitectures [33, 34, 35, 57, 58, 59] or specific domain of applications [54, 60].

Lacking runtime information, however, these parallelization techniques must be conservative and consider the large number of possible paths of program execution, some of which may be rarely executed. This requires the compiler to respect control and data dependencies that do not appear in the actual execution path, inhibiting parallelization and requiring highly accurate alias analysis. On the other hand, with full access to source code, these techniques have the potential to extract coarse-grained parallelism that typically produces the largest speedups and is also most appropriate on certain architectures which have high inter-core communication overhead.

Ever since the multiscalar architecture [61], thread-level speculation has been used to release spurious dependency constraints caused by conservative static analysis [62, 63, 64, 65, 66, 67, 68, 69, 70]. Thread-level speculation allows the compiler to automatically parallelize a program in the presence of statically ambiguous data dependencies, thus effectively extracting parallelism if dynamic dependencies actually do not exist at run time. Such speculative execution usually requires hardware support to detect data dependency violations at run time, which involves comparing load and store addresses that may have occurred out of order with respect to sequential execution. For example, [67] leverages invalidation-based cache coherence for mis-speculation detection. The basic design is to extend those existing invalidation messages to detect data dependency violations by noticing whenever an invalidation arrives from any logically earlier thread for a cache line that has been speculatively loaded by any logically later thread. However, high speculation accuracy typically requires heavy programmer annotation or comprehensive profiling, both of which can be difficult in practice. When neither requirement is satisfied, the achieved speedups are unimpressive at best and the dynamic length of speculated threads is only several hundred instructions, which are hard to expose very useful coarse-grained parallelism [65].

2.4.2 Static Binary Parallelization

Static parallelization techniques [71, 72, 73] have also been enhanced to analyze and transform binary code directly using the binary rewriting technology [74]. All these techniques analyze the binary executable and reconstruct data structures and control flows that were present in the high-level source code. While Vizer [71] performs loop vectorization, two other techniques [72, 73] focus on loop parallelization. For example, [72] extracts address expressions using simple pattern matching to recognize counter initialization, test, and increment for affine loops. It also uses several simple dependency tests to decide whether the given loop is worth parallelizing. These dependency tests are directly adopted from affine loop parallelization at the source code level. As a step further, [73] captures the data flow of address computations, and uses symbolic analysis to reconstruct address expressions built around normalized loop counters. Furthermore, it leverages the polyhedral model to parallelize affine loops, which is fundamentally superior to simple dependency tests.

Due to the difficulty of decompilation, however, these parallelization techniques have extremely limited applicability. Vizer [71] is only applied to three benchmarks, all of which are simple scientific applications, such as 3D tridiagonal solver, matrix addition, and multiplication. Similarly, both [72] and [73] are only used to parallelize affine loops in kernel applications, including the PolyBench benchmark suite and the Stream benchmark suite. No attempts have ever been made to parallelize more irregular programs that dominate everyday use. As stated in [73], any tiny irregularity in the candidate loop can make it unparallelizable, such as the presence of function calls and various loop optimizations (e.g., loop tiling, which produces non-strictly linear address expressions) that are usually performed by the compiler by default.

2.4.3 Dynamic Source Parallelization

A variety of dynamic parallelization techniques have been proposed to insert control logic into source code at compile time, which is used to select the best parallelization strategy at run time. For example, just-in-time scheduling [75] dynamically selects the most beneficial chunk sizes for LLP, and Merge [76] tolerates changing hardware by building separate function versions for different accelerators. Although these techniques can adapt to the program behavior at run time to some extent, the basic parallelization strategy is predetermined at compile time and cannot be updated after the program starts running. Furthermore, these techniques still cannot handle dynamically loaded libraries and self-modifying code, which are always in binary formats.

Some prior work has argued for a JVM-like layer to dynamically optimize and parallelize programs [77, 78]. The major insight is that applications cannot be parallelized just once. They require separate parallelization targeted to the actual hardware and execution environment upon which they will run. However, such techniques assume a dominant programming language such as Java, which has relatively small applicability.

2.4.4 Dynamic Binary Parallelization

Surprisingly, DBP techniques have actually been existing for several decades, long before the many-core era that is just upon us. The superscalar architecture [79] is the most successful parallelization technique that exploits ILP transparently from the binary executable. Ever since the mid-1990's, OoO superscalar execution has been dominating the microarchitecture market. This technology *transparently* parallelizes all kinds of software that is widely used in research, industry, and more importantly, everyday life. On the other hand, parallel programming languages [80] and compilers [56] that can extract LLP and TLP have also been available for decades, but have primarily been used for high-performance computing (HPC) applications.

History has shown that the HPC community will use all kinds of parallelization techniques to exploit the latest advances in many-core architectures that start to dominate the microarchitecture market. However, there is no precedent to show that non-transparent techniques will be adopted for mainstream computing, and in fact there are many reasons to believe otherwise. Some companies will undoubtedly continue to produce sequential programs due to the high cost of porting existing software, updating tool chains, and re-training employees. Furthermore, many legacy programs will be difficult or impossible to update with non-transparent techniques because source code is not available. Finally, transparency is important for portability and forward compatibility of the program, as the range and diversity of many-core architectures grow. These and other factors may be important enough that non-transparent techniques are not adopted in the mainstream, despite the lack of viable alternatives. The quality of transparent parallelization may be the limiting factor on the impact of many-core architectures on mainstream computing.

Several extensions have been made to the superscalar architecture for finding more ILP, including trace processing [25], dynamic multi-threading [22], and speculative multi-threading [23, 24]. These systems use hardware-only technologies to speculate multiple very short threads and execute them simultaneously on different functional units. In order to achieve reasonable speculation accuracy, however, these systems construct very short traces, which necessitates ultra-low communication la-

tency to support program state transfer. Furthermore, these systems rely on fine-grained selective recovery from frequent mis-speculation. While suitable for simultaneous-multithreaded or clustered microarchitectures, neither requirement can be easily satisfied on many-core architectures. More recently, similar technologies have been adopted by several Java virtual machines to extract parallelism from DOALL loops and recursive functions [81, 82, 83].

Several DBP technologies have also been proposed to support many-core architectures, which are generally divided into two main categories: parallelizing the raw DIS [3, 4, 5] and parallelizing the dynamically-generated CFG [6, 7, 8, 9, 10]. DIS-based techniques use extra hardware to combine multiple cores to work cooperatively as a wider core. Native OoO execution could also be considered as a DIS-based technique, which has been widely adopted by modern microarchitectures. Focusing on exploiting ILP, this technology has wide applicability, because ILP typically exists throughout the entire program (with different amounts). Limited by branch prediction accuracy and instruction window size, however, this technology generally fails to exploit distant or coarse-grained parallelism, resulting in relatively mediocre speedups. On the other hand, CFG-based techniques expose a global view of the program and allow discovery of more coarse-grained LLP and TLP, which have the potential to produce much larger speedups. However, analysis on the CFG must be conservative and consider the large number of possible paths of program execution, some of which may be rarely executed. This requires the compiler to respect control and data dependencies that do not appear in the actual execution path, inhibiting parallelization and requiring extensive speculation. When source code is not available, this problem is exaggerated due to the lack of high-level information (e.g., types, variables, data structures), which is essential to achieve accurate alias analysis. Furthermore, when coarse-grained parallelism is hard to exploit, CFG-based techniques typically lack the capability to extract fine-grained parallelism instead and just execute the program sequentially. Thus, it is not surprising that most of the existing CFG-based techniques [6, 7, 10] have failed to parallelize at least half applications in the selected benchmark suite.

Although DIS- and CFG-based techniques are complementary to each other, no prior research has tried to implement both technologies under a unified system. Such a system may achieve large speedups from code regions that contain coarse-grained LLP or TLP, and exploit ILP from the remainder of the program. This combination is quite vital. As Amdahl's Law shows, even a small fraction of non-parallelizable code can drastically inhibit overall speedups. Asymmetric architectures [11, 12] try to address this problem by providing one or more large, OoO cores on the chip for sequential execution modes, and a larger number of simple, IO cores to maximize throughput. Due to chip area constraints, however, this organization reduces the number of total

cores on the chip, hurting the performance of code regions where plentiful LLP can be exploited. The OoO core may also not be available when needed, if multiple applications are sharing the CPU. Furthermore, when LLP or TLP is limited, there will often be more coarse-grained tasks needing further acceleration via ILP than there are OoO cores. Finally, prior research [3, 4, 5] suggests that cooperative work of multiple IO cores can generally outperform or at least compete with an OoO core, which may obviate the need for an asymmetric architecture in the first place.

Chapter 3

Limit Study on Parallelizing Traces

Before developing any specific design of trace-based DBP, it is both important and necessary to conduct a limit study to prove its feasibility. Our limit study has two goals: 1) identify the performance limits of trace-based DBP, and 2) explain *why* trace-based DBP performs as it does. The first goal is to set up the performance upper bound so that any following specific design can be judged to determine whether it has fully exploited the benefits of trace-based DBP. The second goal is to identify the unique and powerful characteristics of trace-based DBP that enable it to achieve substantial speedups. We highlight these characteristics by comparing trace-based DBP to static parallelization which is typically performed by the compiler.

3.1 Overcoming Inherent Handicaps of Static Parallelization

We expect trace-based DBP to overcome two inherent handicaps of static, compile-time parallelization. First, trace-based DBP can exploit parallelism across boundaries between application and library code because traces are naturally formed in which application and library instructions are interleaved. This is not possible at compile time because dynamically loaded libraries are not available. Even for those libraries that are accessible at compile time, they are normally in binary formats and cannot be handled by most static parallelization techniques, all of which require source code to be analyzed. Plugins and dynamically-generated code usually cause the similar problem.

Second, trace-based DBP can perform more aggressive parallelization than static techniques because every trace typically has fewer true dependencies than the CFG. This is because analysis of the CFG will produce true dependencies for all possible execution paths, while analysis of each trace will produce true dependencies for only a single execution path which is actually taken. For

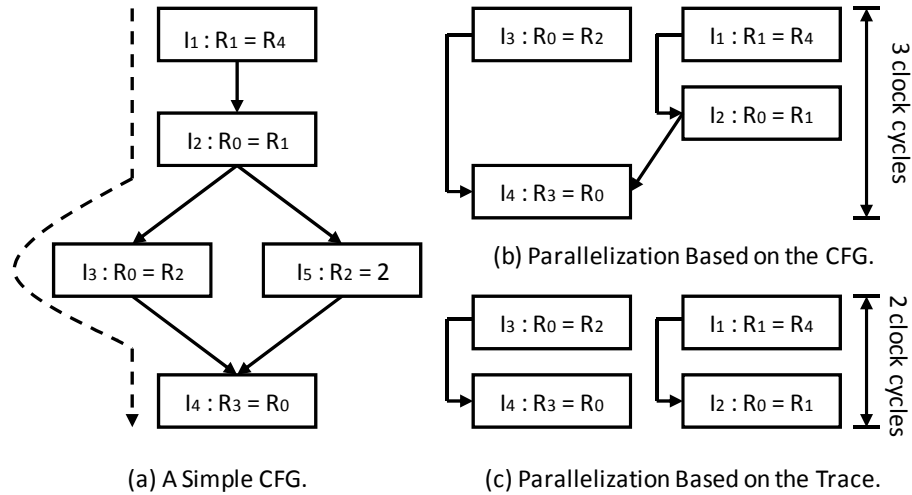


Figure 3.1: Analysis of the trace (dashed arrow) produces fewer true dependencies than analysis of the CFG, leading to improved parallel performance.

example, Figure 3.1(a) shows the CFG of a small program snippet containing five instructions: I_1 , I_2 , I_3 , I_4 , and I_5 . Analysis of this CFG reveals three true dependencies: $I_1 \rightarrow I_2$, $I_3 \rightarrow I_4$, and $I_2 \rightarrow I_4$. The last true dependency exists due to the possible execution path through I_5 . The best possible parallelization of the left branch (dashed arrow) in this CFG that respects all three true dependencies is depicted in Figure 3.1(b), with a parallel execution time of three clock cycles. On the contrary, if the execution path along the left branch is converted into a trace at run time, an analysis of the trace would not find the true dependency $I_2 \rightarrow I_4$ because I_3 produces the most recent value of R_0 . A parallelization of this trace would thus run the same instructions with a parallel execution time of only two clock cycles, as depicted in Figure 3.1(c). Thus, trace-based DBP can be more aggressive than static parallelization because it must only consider true dependencies that manifest in a single execution path, while static techniques tend to be conservative because they respect true dependencies from all possible execution paths.

3.2 Limit Study Setup

We analyze the performance limits of trace-based DBP by making three idealizations about the hardware and algorithms: 1) the program runs on a many-core architecture with an unbounded number of cores and an unlimited, shared L1 cache, 2) the trace construction algorithm can always identify the most frequently repeating patterns of instructions that will occur in a particular run of the program, and 3) when the program reaches a repeating trace, the trace prediction algorithm can always correctly predict the trace that is about to run, which we call *perfect one-step prediction*.

These three idealizations are reasonable in the sense that real hardware and algorithms should actually approach them as they improve over time. If the number of cores double every 18 months as expected [1], processors may soon have more cores than trace-based DBP could ever use. The trace construction algorithm will improve dramatically as it is increasingly informed by compile-time analysis [84] and program profiling [85, 86]. When the next executed trace has been cached, perfect one-step prediction is already possible by simply executing in parallel all existing traces that begin with the next target address and, from this perspective, improvements in trace prediction will simply reduce the number of cores required for perfect one-step prediction. Thus, we expect the performance upper bound found in this limit study to be tight in the sense that the performance of trace-based DBP may actually approach this bound as technologies evolve.

In order to conduct this limit study, we perform a four step process for each application in the SPEC CPU2000 benchmark suite: 1) record the complete execution sequence of the program, 2) analyze the recording offline to identify the frequently repeating traces, 3) create a new execution sequence by replacing each trace in the original execution sequence with the parallelized version, and 4) analyze the parallel execution time of the new execution sequence using a model of a shared-memory many-core architecture. The *perlbmk* benchmark was omitted because it recursively calls itself, starting multiple instances of our capture framework and exhausting memory of the machine. All benchmarks are executed using the test data sets as input. In the following sections, we will describe in detail how we implement each of these steps.

3.2.1 Recording Execution Sequences

We record the original execution sequence of the program by inserting instrumentation code to the binary executable. This is performed by employing translation-based dynamic instrumentation to the benchmark during its execution. Whenever a new basic block is translated, instrumentation code is inserted to record the program counter whenever the basic block gets executed. Instrumentation code is also inserted to record the effective address of every load and store instruction, which are used to perform perfect memory disambiguation, as will be described in Section 3.2.3.

The instrumentation code saves necessary state of the program, calls the appropriate logging code, and then restores the state of the program before continuing execution. All instrumentation is performed at the binary code level and not at the source code level. This avoids unintended interactions between instrumentation code and compiler optimizations, thereby ensuring that we are executing the true binary executable for each benchmark.

Recording the complete execution sequence of the program produces a large amount of information and so we use double buffering [87] to reduce runtime overhead and apply the VPC3 algorithm [88] to compress the collected information. This greatly increases execution speed and reduces disk space requirements, although a typical one second program still requires three minutes and fifty megabytes of disk space to record. We record program execution on a SPARC/Solaris machine, in part because RISC ISAs are understood to be more suitable for many-core architectures [89]. The results of our limit study should generalize at least to other RISC ISAs.

Prior research has also built systems to efficiently record program execution [85, 86], and some can even replay the recording by capturing non-deterministic events such as interrupts, preemption, and user input [90, 91]. Like these systems, our capture-analysis framework can be widely applicable to future studies on dynamic trace-based systems.

3.2.2 Analyzing Execution Sequences to Construct Repeating Traces

Once the execution sequence has been recorded, we construct traces by finding all frequently repeating patterns of instructions. We do this using an offline dictionary-based algorithm that is typically used for compression [92], shown in Figure 3.2(a). Initially every basic block is defined to be a unique symbol. We then identify the two symbols s_i and s_j that are the most frequent pair of adjacent symbols in the entire execution sequence (lines 2 to 8). If no pair appears more than once, the algorithm stops (line 13). Otherwise, we replace all occurrences of $s_i s_j$ with a new symbol A_j (lines 10 to 11). The execution sequence now has fewer symbols and the algorithm repeats to again find the most frequent pair of adjacent symbols. When the algorithm completes, all symbols remaining on the execution sequence become the selected traces. Figure 3.2(b) shows an example of how traces are constructed on an execution sequence of eight basic blocks (a, b, c, a, b, a, b, c). In the first iteration, ab is found to occur most frequently (three times) and is replaced by a new symbol A . In the second iteration, Ac occurs two times and is replaced by a new symbol B . After that, no pair of adjacent symbols occurs more than once and the algorithm completes, constructing two different traces A (basic block sequence a, b) and B (basic block sequence a, b, c).

The traces constructed in this phase are the ones that will be parallelized in the next phase. The choice of this trace construction algorithm corresponds to the idealized assumption that frequently repeating traces can always be identified at run time, perhaps with the help of compile-time analysis [84] and program profiling [85, 86]. One advantage of this assumption is that a small set of repeating traces cover a large portion of dynamic instructions and thus have a high probability to be

```

Algorithm construct_trace : path
01 loop do
02   for each pair of adjacent symbols (s1, s2)i in path do
03     if check_pair (s1, s2)i then begin
04       numi ← occurrence number of (s1, s2)i
05     end
06   done
07   (s1, s2)m ← most frequent pair
08   freqm ← maximum occurrence number
09   if freqm > 1 then begin
10     Aj ← create new symbol
11     replace all occurrences of (s1, s2)m with Aj
12   end else begin
13     break
14   end
15 done

```

(a) Idealized Trace Construction Algorithm.

	execution path	pair (max. num.)	new symbol
1	path → abcababc	ab (3)	A → ab
2	path → AcAAc	Ac (2)	B → Ac
3	path → BAB		

(b) Example of Trace Construction.

```

Algorithm check_pair : (s1, s2)i

```

```

01 return true

```

(c) No Handicap.

```

Algorithm check_pair : (s1, s2)i

```

```

01 if (s1 ∈ app && s2 ∈ app) || (s1 ∈ lib && s2 ∈ lib) then begin
02   return true
03 end else begin
04   return false
05 end

```

(d) Handicapped Algorithm that Cannot Parallelize across Boundaries between Application and Library Code.

Figure 3.2: The idealized trace construction algorithm finds the most frequently repeating patterns of instructions in the entire execution sequence, as shown in the example. The handicapped version does not construct traces across boundaries between application and library code.

predicted accurately [46]. One study of short traces about seven basic blocks long has demonstrated that even a very shallow execution history is effective enough to achieve close to 90% prediction accuracy on average [93].

We can modify the trace construction algorithm to handicap trace-based DBP so that it cannot parallelize across boundaries between application and library code. More specifically, we replace the original *check_pair* function (Figure 3.2(c)) invoked on line 3 of the trace construction algorithm with an alternative version that only allows two adjacent basic blocks to be combined into a single symbol if both of them belong to application code or both of them belong to library code. The pseudo code for this handicapped algorithm is illustrated in Figures 3.2(d) and its effects on execution speed will be analyzed in Section 3.3.

3.2.3 Parallelizing Execution Sequences

Once the repeating traces in the execution sequence are identified, they are parallelized using a modified version of the dynamic critical path scheduling algorithm [94], which is derived from prior research on allocating task graphs to fully-connected multi-processors. This algorithm is selected because it has been experimentally demonstrated to produce the minimum execution time among all comparable algorithms. For the purposes of trace parallelization, we define each instruction to be a separate task. For each trace, we perform the following four steps:

1. Identify true dependencies in the trace and build the dependency graph. Initialize the current schedule to be an empty schedule.
2. Calculate the absolute earliest start time (AEST) and absolute latest start time (ALST) of each instruction based on the current schedule. Let L be the group of instructions with the smallest value of $ALST - AEST$, and pick instruction i from L that does not have predecessors in L . Ties are broken arbitrarily.
3. Schedule instruction i on core j where 1) after insertion, it does not delay the ALST of all instructions that are already scheduled on that core, including itself, and 2) there are no violations of any true dependencies.
4. Go back to Step 2 if all instructions are not scheduled.

Calculation of the AEST and ALST requires an analysis of the program's dependency graph. True register dependencies are easily identified. Anti and output register dependencies are eliminated using renaming technologies to increase scheduling flexibility. On the other hand, many of

memory dependencies are ambiguous, where two load or store operations refer to a memory address that has not yet been calculated, and so the system cannot tell whether or not there is a dependency between them. We use the actual effective addresses that are recorded during the original execution sequence to disambiguate these memory references, and only take the real memory dependencies into account when calculating the AEST and ALST. Memory disambiguation is a standard technique employed by many parallelization and OoO execution systems to execute memory reference instructions (i.e., loads and stores) out of program order. Both software-based and hardware-based techniques have been developed, and experimental results have shown approximately 80% accuracy in memory disambiguation for some applications [28, 95].

After all the traces are parallelized, we replace their occurrences in the original execution sequence with the parallelized versions. This new execution sequence represents the idealized execution sequence that a trace-based DBP implementation might produce in the real world. Correctly replacing every single trace in the original execution sequence with the parallelized version corresponds to the idealized assumption of perfect one-step prediction. As stated earlier, when the next executed trace has been cached, this assumption can be satisfied if a trace-based DBP system can execute multiple predicted traces in parallel.

We can modify the trace parallelization algorithm by considering more or less data dependencies in Step 1. For example, we can handicap the algorithm by considering all true register dependencies on the CFG, instead of only those in the trace. This would emulate the handicap of static parallelization that is conservative because it has to respect all true dependencies carried by every possible execution path. On the other hand, we can employ optimizations such as constant propagation before parallelization to eliminate unnecessary register dependencies and similarly, we can eliminate all memory dependencies, which would emulate the advantage of having perfect value prediction [96, 97]. The effects of these modified versions of the trace parallelization algorithm on execution speed will be analyzed in Section 3.3.

3.2.4 Modeling Parallel Execution Time

Once the parallel execution sequences are created, we analyze them using a model of a shared-memory multi-core architecture to calculate the parallel execution time. We assume hardware that is ideally suited for trace-based DBP and that is not currently available on the market. However, all individual components and features of our hardware model are either currently available or are achievable or nearly achievable using current technologies. The only aspect of the hardware model

that is not currently achievable is the assumption of an unlimited number of cores with an unbounded, shared L1 cache (i.e., no memory access latency), which becomes one of the idealized assumptions. We use the same hardware model to calculate the time of both the unmodified sequential execution sequence and the parallelized execution sequence.

In order to calculate the execution time of a sequence of instructions, we define each instruction to have an execution time of one clock cycle due to pipelining. We define the execution time of a parallelized trace to be the maximum AEST of all instructions in the trace plus one, to account for the execution of the last instruction. We require at least one clock cycle to separate any two instructions with true register dependencies and any memory dependencies that execute on different cores, for inter-core communication. Software-based synchronization techniques such as locks, barriers, and monitors can cause more than one clock cycle of runtime overhead due to the interactions with the operating system, but special hardware such as synchronization array [98] and operand network [99] provide efficient, non-memory-based communication between different cores on the same chip. These technologies enable the production and consumption of a single register value on different cores to be performed in back-to-back cycles, as long as the path between the two cores is not congested. Thus, we aggressively set the inter-core synchronization time to be one clock cycle.

Dynamic trace-based systems typically incur runtime overhead for analyzing and manipulating traces, inserting new traces into the trace cache, deleting outdated traces from the trace cache, and other trace management operations. On many-core architectures, we assume that all of these operations are shipped to idle cores, and that it has no impact on the execution speed of the actual program. Even for single-core architectures, the total runtime overhead of SDT can be as low as 3% [100, 101], which can be further amortized when used in combination with one of the many other reasons for virtualization.

Dynamic trace-based systems also incur runtime overhead because they have to observe execution patterns in order to construct traces. One widely used technology is software-based dynamic instrumentation in which the system inserts instrumentation code to each basic block for obtaining the execution history. However, this technology can cause the target program to run up to two times slower on average [41, 42]. In the mean time, a variety of hardware-based instrumentation techniques have been proposed to directly integrate trace construction with non-critical stages of instruction pipeline execution, which dramatically reduce runtime overhead. Recently, Intel Research has designed log-based architecture [102] that captures an instruction-grained log from a monitored program and ships it to another core that performs further processing. Additionally, prior research has also proposed a chip-wide branch trace buffer [45], which makes the execution history of one core

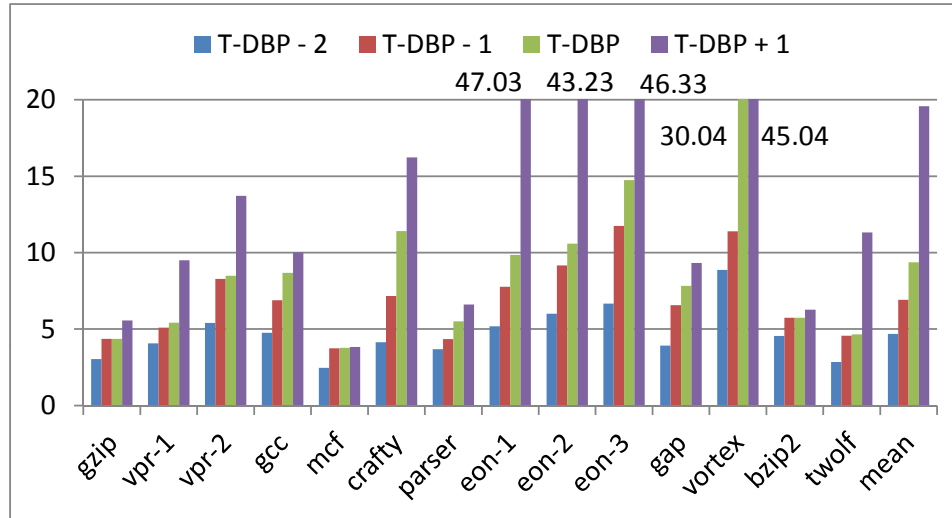
visible in real time to the other cores on the same chip. Use of these transparent, hardware-based techniques would make the runtime overhead of execution monitoring negligible and thus, we assume that trace construction occurs simultaneously with actual program execution and does not need to consume any extra clock cycles.

3.3 Experimental Results

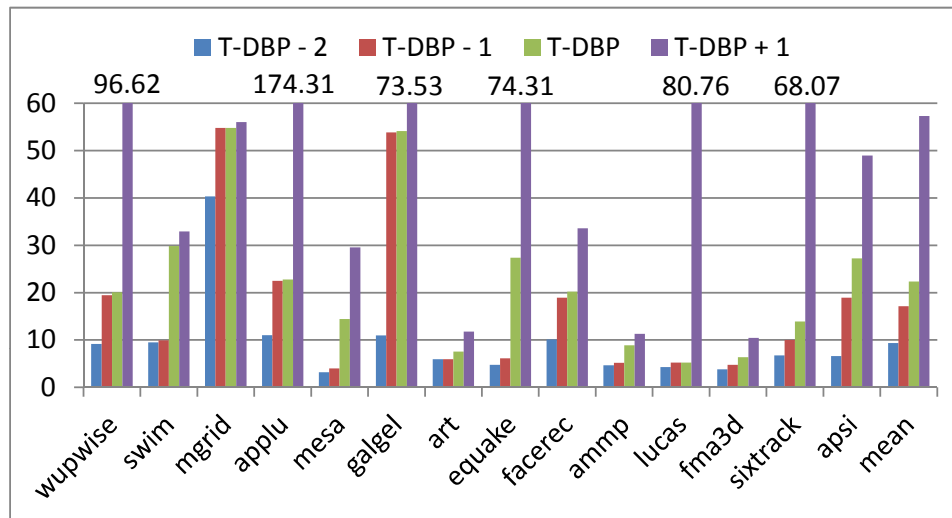
We use the limit study framework described in Section 3.2 to analyze the performance of trace-based DBP on the SPEC CPU2000 benchmark suite. We test and compare four different versions of the trace-based DBP implementation. The first two implementations use handicapped versions of the trace construction and parallelization algorithms, as described in Sections 3.2.2 and 3.2.3, respectively. These handicaps emulate varying degrees of the two handicaps of static parallelization described in Section 3.1. The third implementation is standard trace-based DBP with no handicaps applied. The fourth implementation is an enhanced version of trace-based DBP that uses constant propagation to eliminate unnecessary register dependencies and uses perfect value prediction to eliminate all memory dependencies, as described in Section 3.2.3. These four implementations are named and defined as follows:

- T-DBP-2: trace construction cannot cross boundaries between application and library code; trace parallelization is constrained by all true dependencies found on the CFG.
- T-DBP-1: trace construction cannot cross boundaries between application and library code.
- T-DBP: both trace construction and trace parallelization are unconstrained.
- T-DBP+1: trace parallelization does not consider unnecessary register dependencies and all memory dependencies.

The performance of all four trace-based DBP implementations is illustrated in Figure 3.3, with the results of integer benchmarks and floating point benchmarks put into separate graphs. The average speedup over sequential execution is 9.36x and 22.34x for integer and floating point benchmarks, respectively. This number can be as high as 30.04x for integer benchmarks (*vortex*) and 54.81x for floating point benchmarks (*mgird*). The higher speedup for floating point applications is likely due to the fact that they contain a larger fraction of numerical code, which has shown to introduce fewer true dependencies by prior research.



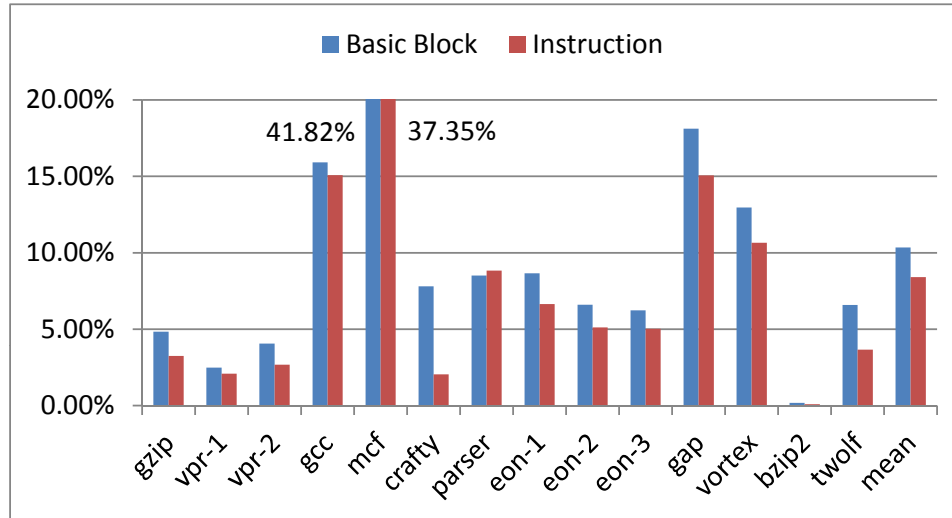
(a) CINT2000.



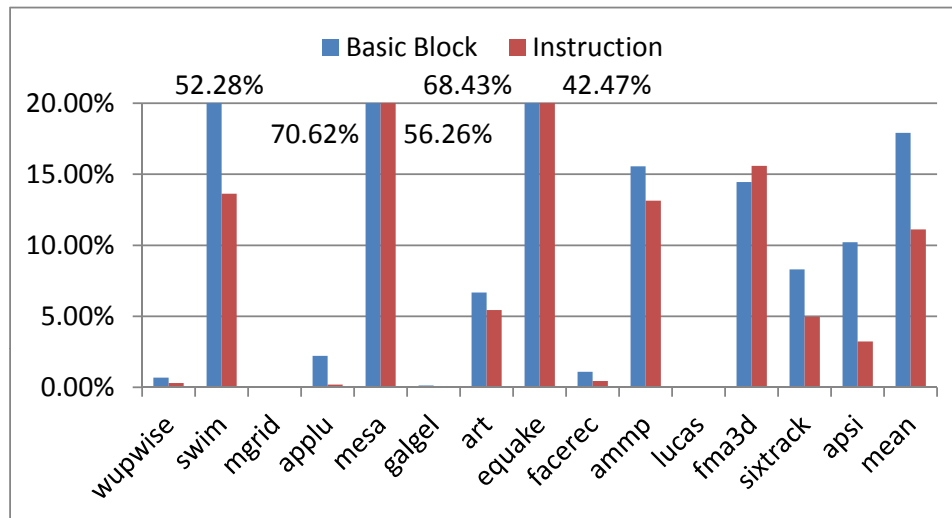
(b) CFP2000.

Figure 3.3: The standard T-DBP achieves an average speedup of 9.36x and 22.34x over sequential execution for integer and floating point benchmarks, respectively. When all handicaps are artificially emulated, the average speedup shrinks to 4.68x for integer benchmarks and 9.36x for floating point benchmarks. Traditional constant propagation and perfect value prediction could potentially improve execution speed by another factor of 1.97x for integer benchmarks and 3.49x for floating point benchmarks on average.

When all handicaps are emulated, the average speedup over sequential execution is 4.68x for integer benchmarks and 9.36x for floating point benchmarks. This supports the hypothesis that the ability of trace-based DBP to overcome these two handicaps accounts for its ability to explore a high degree of parallelism. Eliminating these handicaps improves the performance of trace-based DBP by more than two times. In the following sections, we will analyze in detail the degree to which various aspects of trace-based DBP affect its performance.



(a) CINT2000.

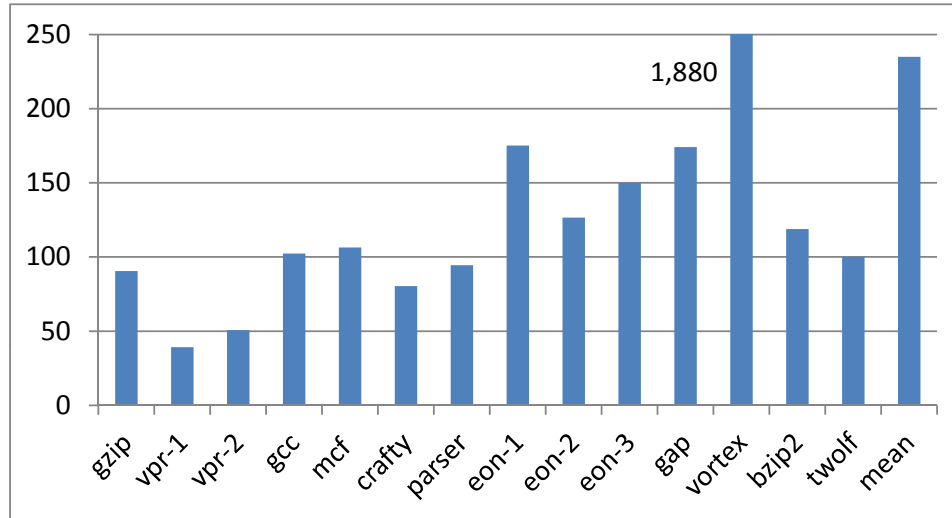


(b) CFP2000.

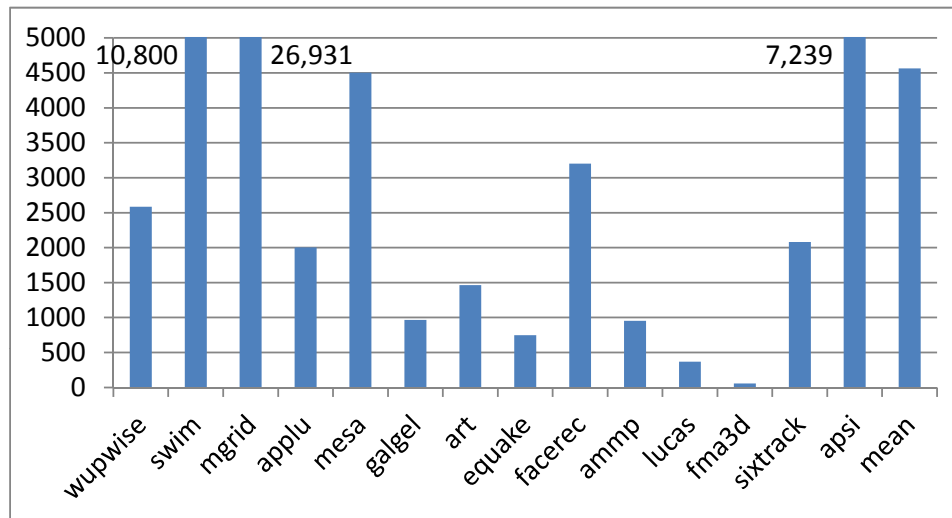
Figure 3.4: An average of 10.32% basic blocks or 8.40% instructions for integer benchmarks and 17.91% basic blocks or 11.12% instructions for floating point benchmarks belong to libraries.

3.3.1 Analysis of Trace Construction

The relative results of T-DBP-1 and T-DBP indicate that parallelizing across boundaries between application and library code can improve the average speedup over sequential execution from 6.91x and 17.12x to 9.36x and 22.34x, for integer and floating point benchmarks, respectively. Note that the speedup does not necessarily correspond to the percentage of executed basic blocks or instructions that are from libraries, which is illustrated in Figure 3.4. In fact, *mcf* executes more library instructions than all other integer benchmarks but also shows the minimum improvement between T-DBP-1 and T-DBP. On the other hand, the performance of *crafty* and *vortex* degrades



(a) CINT2000.

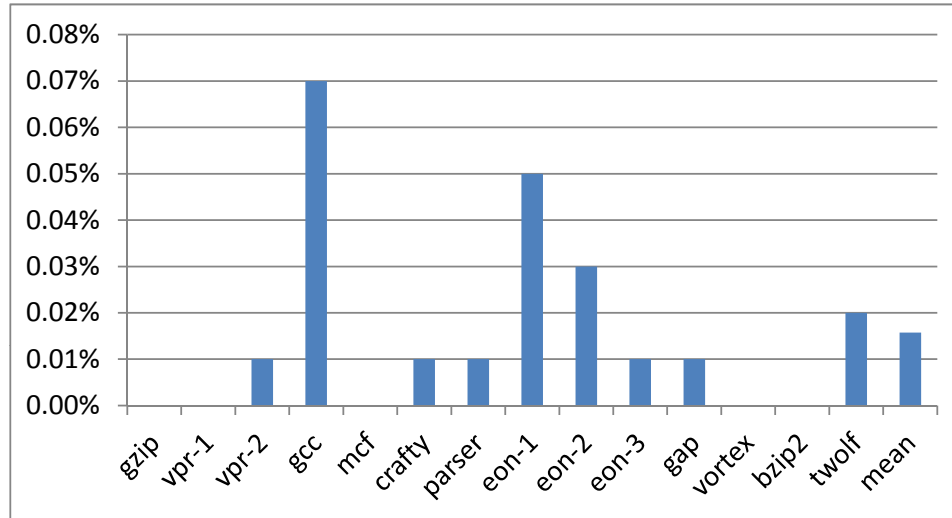


(b) CFP2000.

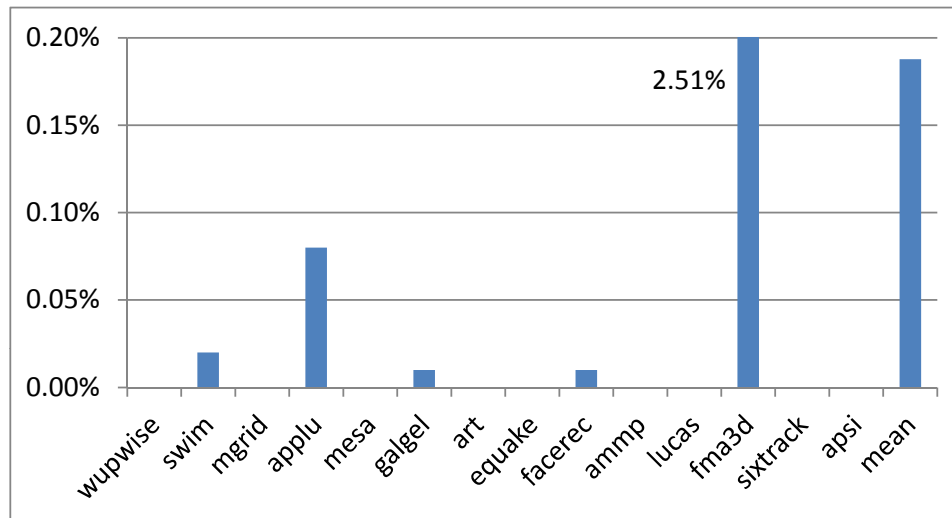
Figure 3.5: The average trace length is 235 basic blocks for integer benchmarks and 4,565 basic blocks for floating point benchmarks.

dramatically with only 2% and 11% of its instructions executed in libraries, respectively. Also note that the library instructions are being parallelized in all versions of the trace-based DBP implementation. The handicapped version only eliminates parallelization *between* application and library instructions. The degree to which this handicap affects the speedup is related to the degree to which application instructions are interleaved with library instructions. These results validate the hypothesis that the inability to parallelize across boundaries between application and library code is a significant handicap for static parallelization.

When all handicaps are removed, trace-based DBP constructs very long traces. Figure 3.5 depicts



(a) CINT2000.



(b) CFP2000.

Figure 3.6: An average of 0.02% basic blocks for integer benchmarks and 0.19% basic blocks for floating point benchmarks are not formed into traces.

the average number of basic blocks within each constructed trace, which can be as large as 1,880 for integer benchmarks (*vortex*) and 26,931 for floating point benchmarks (*mgrid*). This indicates that the applications in this benchmark suite tend to repeat long sequences of instructions. Figure 3.6 illustrates the percentage of basic blocks in the entire execution sequence that are not formed into traces. For all applications, nearly all basic blocks are combined to construct longer traces and can be parallelized for later reuse. The singleton basic blocks that do occur are primarily from the prologue and epilogue of the program. Both of these results support the hypothesis that for a typical program, a relatively small number of traces can almost cover all dynamic instructions,

which suggests good trace predictability.

3.3.2 Analysis of Trace Parallelization

The only difference between T-DBP-2 and T-DBP-1 is that T-DBP-2 performs dependency analysis on the CFG during the parallelization process while T-DBP-1 performs dependency analysis on traces. Thus, the relative results of these two versions of the trace-based DBP implementation indicate the degree to which parallelism increases when performing dependency analysis on traces at run time instead of on the CFG at compile time. The average speedup of T-DBP-2 over sequential execution is 4.68x and 9.36x for integer and floating point benchmarks, respectively, while it is 6.91x and 17.12x of T-DBP-1. Thus, in this experimental condition, dependency analysis on traces can produce an average speedup of 1.47x for integer benchmarks and 1.77x for floating point benchmarks over dependency analysis on the CFG. These results validate the hypothesis that dependency analysis on the CFG is a significant handicap for static parallelization.

3.3.3 Analysis of Constant Propagation and Value Prediction

The relative results of T-DBP and T-DBP+1 indicate the degree to which dynamic optimizations such as constant propagation and value prediction could possibly improve the performance of trace-based DBP. The results indicate that applying traditional constant propagation on traces before parallelization and applying perfect value prediction during dependency analysis would improve execution speed by an average factor of 1.97x and 3.49x over standard trace-based DBP, for integer and floating point benchmarks, respectively, and can improve the final speed by up to 174.30x (*applu*) over sequential execution. These results approximate the absolute limit of trace-based DBP, when unnecessary register dependencies are eliminated and all memory values can be predicted in advance. State-of-the-art value prediction techniques can achieve approximately 90% prediction accuracy on some applications [97, 103], although we expect that value prediction accuracy may be higher with trace-based DBP, which could enable the development of new value prediction techniques that leverage runtime information about traces.

3.4 Summary

In this chapter, we study the performance limits of trace-based DBP by making three idealized assumptions: 1) an unlimited number of cores and an unbounded amount of shared L1 cache, 2) the

construction of most frequently repeating traces at run time, and 3) perfect one-step prediction of the trace that is about to run. Our results demonstrate the ability of trace-based DBP to produce an average speedup of 9.36x and 22.34x for integer and floating point benchmarks, respectively. We hypothesize that this improvement is due to the ability of trace-based DBP to overcome two key handicaps of static parallelization: 1) it cannot parallelize across boundaries between application and library code, and 2) analysis of the CFG identifies true dependencies that do not actually appear during program execution. We quantify the effects of each of these handicaps by artificially applying them to trace-based DBP, and show that when all handicaps are emulated, the average speedup does indeed drop dramatically to 4.68x for integer benchmarks and 9.36x for floating point benchmarks. On the other hand, dynamic optimizations such as constant propagation and value prediction could potentially improve execution speed by another factor of 1.97x for integer benchmarks and 3.49x for floating point benchmarks on average.

Trace-based DBP is not an alternative to most other parallelization techniques. It is an orthogonal technique that can be applied simultaneously to create a multiplicative gain. For example, trace-based DBP can be combined with value prediction to remove memory dependencies from traces [7, 68], and can be applied to each thread created by manual parallelization techniques [51, 49, 48, 50, 52]. Trace-based DBP can also be combined with other program manipulations that require program virtualization and/or dynamic binary translation, such as optimization [14, 15, 18, 20, 21], ISA translation [26, 27], profiling [41, 42], or security monitoring [43, 44]. Combining such operations would amortize any overhead of the virtual execution engine.

Chapter 4

The Tracy Framework

This chapter presents *Tracy*, an innovative DBP technology that explores the possibility of leveraging hot traces to provide a large instruction window without introducing spurious dependencies. Tracy monitors a program at run time and dynamically identifies these hot traces, parallelizes them, and caches them for later use so that the program can run in parallel every time a hot trace repeats. Inspired by multi-path execution [104, 105, 106], Tracy exploits the unique power of many-core architectures by launching multiple traces and executing them simultaneously on idle cores. The major insight is that in many cases, speculation accuracy can be *dramatically* increased by only trying a *very small* number of candidate traces.

Figure 4.1 illustrates the overall framework of Tracy, which involves five algorithmic components. A conceptual overview of Tracy is explained in Figure 4.2. In Figure 4.2, Core 1 is instrumented with trace management functionality and starts to execute the unmodified, sequential binary. Simultaneously, the *trace constructor* monitors the instruction stream and identifies traces from frequently repeating instruction sequences. The traces are then processed by the *trace parallelizer* and stored in the *trace cache*. This parallelization process is offloaded to spare cores in order not to affect the sequential execution. At every point during execution, the *trace predictor* checks for *candidate traces*: parallelized traces in the trace cache that 1) begin with the instruction that is about to be executed by the sequential binary, and 2) have a high probability of running to completion. If any exist, it suspends the sequential execution and launches them on the remaining available cores (Cores 2 to 7). The speculated traces operate on *copies* of the actual program state. If a trace deviates from the execution path which is actually taken, it aborts and its copy of program state is discarded. If any traces run to completion, one of them is selected and its copy of program state is committed

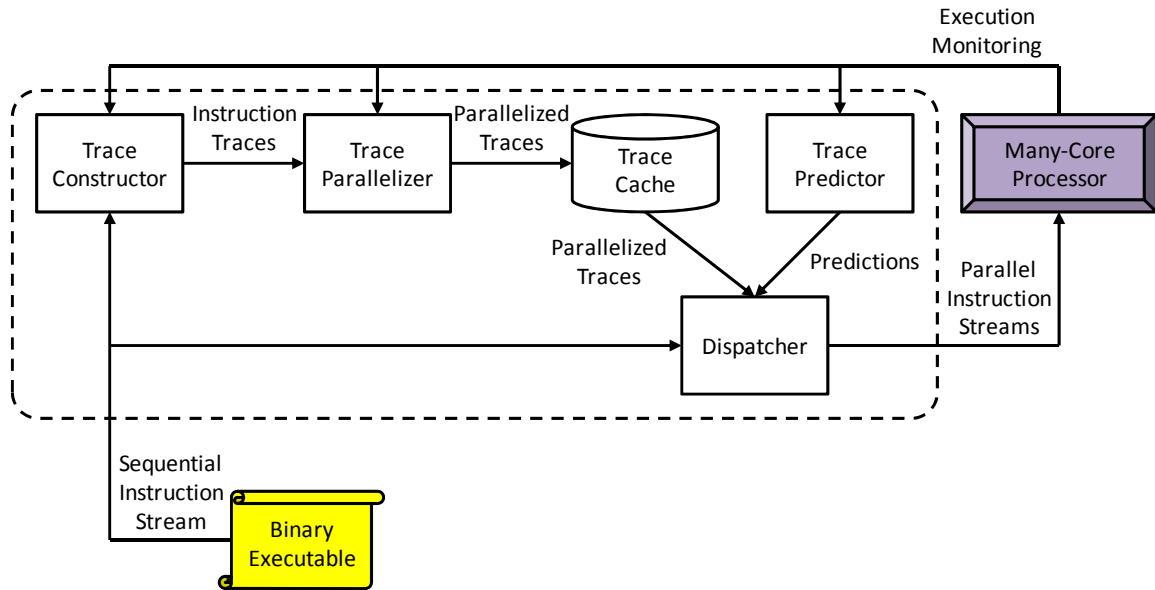


Figure 4.1: Tracy takes a sequential instruction stream and transparently converts it to a set of parallel instruction streams.

to the suspended sequential execution, which “skips forward” in time to the end of the selected trace. Figure 4.2 illustrates three example scenarios. First, the right trace aborts and the left trace succeeds, causing the sequential execution to skip forward. Second, both traces abort and so the sequential binary continues running from the last dispatch point. Third, both traces succeed (it is possible when one trace happens to be the prefix of the other trace) and the copy of program state from the left trace is selected to commit.

4.1 Motivating Example

While it is relatively obvious that long traces expose more parallelism opportunities than the small hardware instruction window used by DIS-based DBP techniques, Figure 4.3 illustrates an example of how Tracy can successfully parallelize some applications when CFG-based DBP techniques fail to. Figure 4.3(a) shows the CFG of a simple loop with a loop-carried dependency on index variable i . Assuming the contents of array x are hard to predict, this dependency serializes every iteration of the loop so that no parallelism can be extracted. During run time, Tracy observes that the program continues repeating two traces (Figures 4.3(b) and 4.3(e)), each of which comprises two iterations of the original loop. (In reality, Tracy would build much longer traces. Traces of two iterations are just for illustrative purposes.) Because the interleaving of these two traces is unpredictable (otherwise, the contents of array x are also predictable), Tracy has to execute both of them simultaneously to

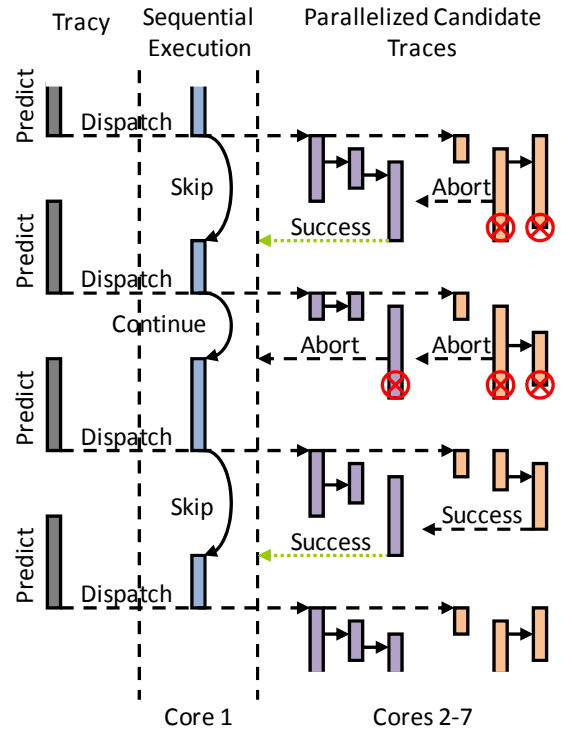


Figure 4.2: Tracy uses one core for trace management plus sequential execution, and the remaining cores for speculative execution of parallelized candidate traces.

achieve high speculation accuracy. Because each trace encapsulates only a single flow of control, control dependencies are eliminated and they can both be aggressively parallelized. Taking the second trace as an example, it is first optimized to eliminate unnecessary data dependencies. As illustrated in Figure 4.3(f), the value of variable i defined at statement 3 is propagated to statements 4, 7, and 8, which breaks the dependency between the two iterations. Thus, these two iterations can run in parallel as depicted by Figure 4.3(g). In Figure 4.3(h), Tracy further performs fine-grained instruction scheduling by moving statements 7 and 8 to the first iteration, achieving better load balance between the two threads.

4.2 Execution Model Justification

Although it dramatically increases speculation accuracy, multi-trace execution does require more cores and possibly more energy that can otherwise be used to execute other applications. In practice, however, this execution model is reasonable and efficient in many computing environments. First, in most server systems for high performance and research computing, once a job is scheduled, it has dedicated use of the cores to which it was allocated. Modern applications usually contain both

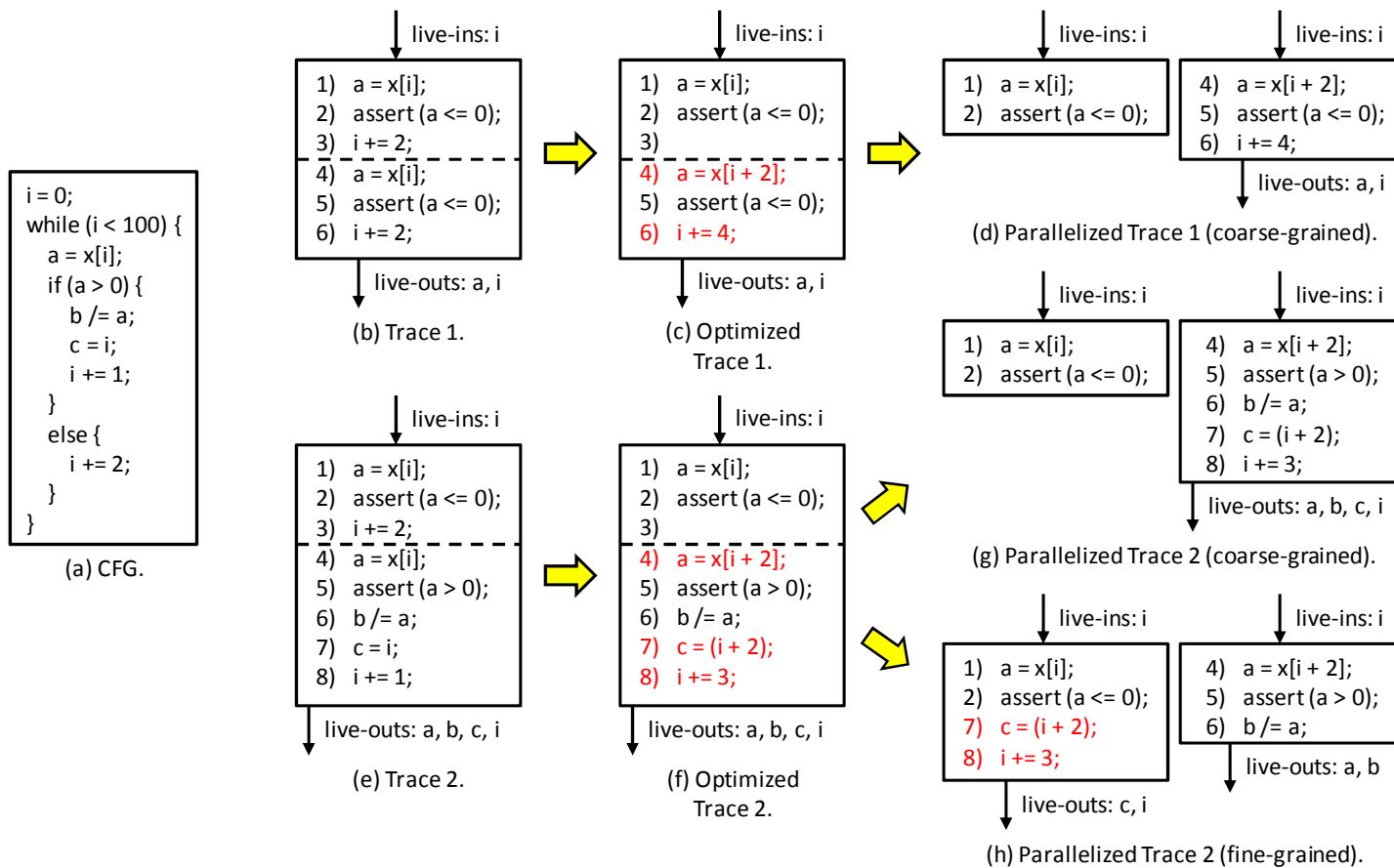


Figure 4.3: Based on multi-trace execution, Tracy can successfully parallelize some applications when CFG-based DBP techniques fail to.

sequential and parallel code regions, and users tend to request resources greedily to support the most parallel parts of the program. These otherwise “wasted” cores can be readily leveraged by Tracy to automatically parallelize the sequential parts of the program. Furthermore, these computing environments do not impose any cost to users for energy, and they are often more concerned in the provisioning stage to ensure sufficient power delivery under peak load. For users interested in getting their results as quickly as possible, there is no disincentive, and every incentive, to use Tracy. Second, the same dedicated-resource policy also applies to datacenters to ensure quality of service (QoS) [107]. Co-location on many-core machines is disallowed for user-facing and latency-sensitive applications to avoid potential interference. As more computing moves into the cloud, these applications are likely to dominate everyday computing in the future. Tracy, on the contrary, can exploit these idle cores to provide better QoS. Energy consumption does matter in datacenters because profit is heavily affected by operating efficiency. However, the CPU chip is only a small fraction of overall energy consumption. Large DRAM, disk, power supply inefficiencies, etc. are also major factors that impose a constant background “system leakage” while the machine is awake. “Wasting” some energy on the CPU chip to complete the task further and thus reducing energy spent on these other factors may be justifiable. For example, if the other factors are constant and the CPU chip consumes 30% of the total power, increasing the CPU chip power by 2x but reducing execution time by even a mere 1.3x is break-even on energy.

4.3 Hardware Architecture

Tracy assumes that a many-core chip is organized into *master clusters* and *slave clusters*, as illustrated in Figure 4.4. Cores within each cluster are connected via a *cluster bus*, and different clusters are connected via a *backbone bus*. Such a hierarchical bus design is inspired by [108], and can be easily replaced by other on-chip networks (e.g., 2D Mesh [109]). The backbone bus is also segmented and connected with simple tri-state gates to pipeline sequential transfer of bulk data between the same source and destination core [108]. In each master cluster, at least one core is instrumented with special hardware to support execution monitoring and trace management, which will be described in Chapter 5. Several other cores are dedicated for trace parallelization, depending on the actual workload during run time. Tracy does not rely on any centralized hardware to support collaborative fetch, renaming, memory disambiguation, or commit, which is essential in existing DBP technologies to enable fine-grained multithreading [3, 4, 5]. Besides trace construction and prediction, however,

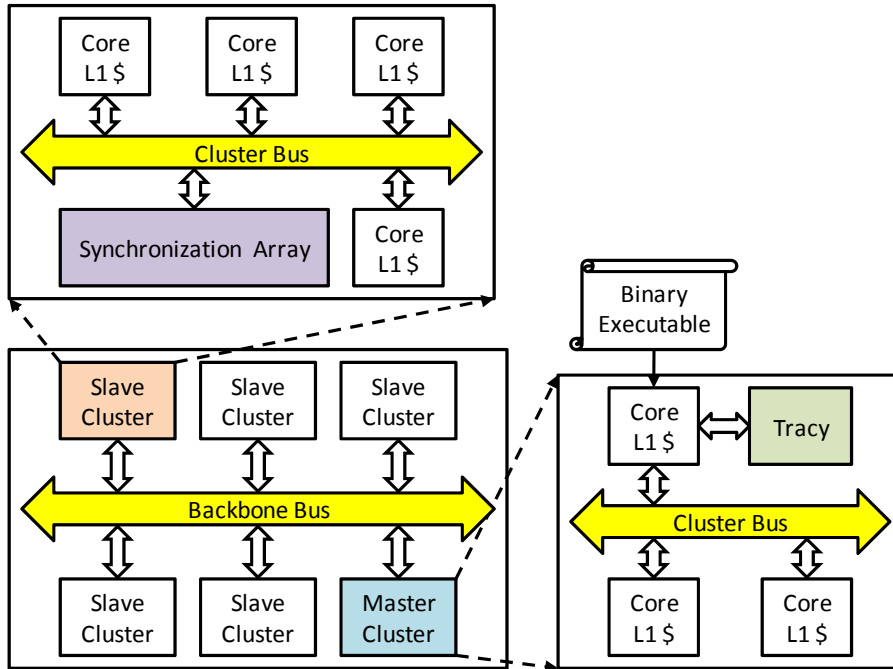


Figure 4.4: Tracy assumes that a many-core chip is organized into master clusters with a specially-instrumented core and slave clusters that contain synchronization arrays.

Tracy does require some extra support from the underlying architecture, which will be described in the following sections in detail.

4.3.1 Supporting Low-Latency Intra-Cluster Communication

Because one important functionality of Tracy is to exploit fine-grained parallelism (e.g., ILP), it requires low-latency communication channels among different cores on the same chip to transfer scalar values. Software-based synchronization techniques such as locks, barriers, and monitors can cause large runtime overhead due to the interactions with the operating system, but special hardware such as synchronization array [98] and operand network [99] provide efficient, non-memory-based communication between different cores on the same chip.

Within each slave cluster, a synchronization array [98] is also connected to all the cores to provide low-latency communication via dedicated links, which are depicted as part of the cluster bus. Each separate link connects one core to the array and does not interfere with cache traffic. As illustrated in Figure 4.5, each entry in the array is correlated to a unique register or memory reference that needs to be synchronized. Entry reuse is not allowed since accesses are totally asynchronous. During program execution, extra *produce* and *consume* instructions inserted in each parallelized thread are used to copy data to and from the array. While consume instructions can run speculatively, the

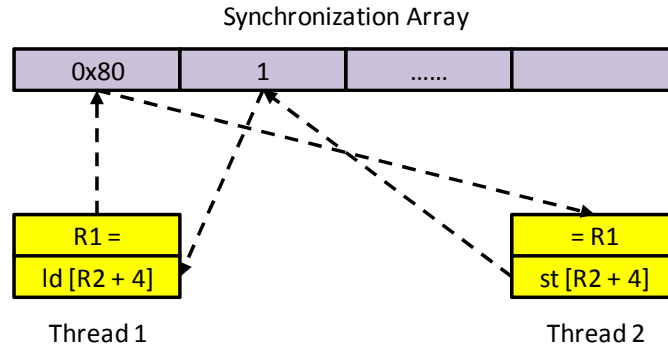


Figure 4.5: Each entry in the synchronization array is correlated to a unique register or memory reference that needs to be synchronized. While the actual value of each synchronized register is explicitly transferred through the array, a boolean value is just enough for a pair of dependent memory references to maintain the correct order.

copy back must happen at the retire stage of the produce instruction. In the current design, after the produce instruction has been decoded, the processor extracts the entry number of the array (no address calculation is needed as in normal memory loads and stores) and stores it with the instruction that actually produces the value. When that instruction retires, the value is directly routed to the array. The hardware cost for this architectural optimization is quite small. For example, in OoO cores, only some extra bits are needed for each ROB entry. If the consumer arrives earlier than the producer, the core number of the consumer is recorded and the data will be automatically redirected to it when the producer finally arrives. While the actual value (e.g., 0x80 in the first entry) of each synchronized register is explicitly transferred through the array, a boolean value (e.g., 1 in the second entry) is just enough for a pair of dependent memory references to maintain the correct order. In order to eliminate unnecessary synchronizations, a partial instruction retire order is constructed by considering both instruction dependencies among different cores and instruction order on the same core. Thus, if one memory reference instruction is dependent on two memory reference instructions that are comparable according to the partial order, it only needs to be synchronized with the one that will retire last. If an instruction needs to consume multiple values, they are encoded into one variable-length instruction. In OoO execution, a separate issue queue (with its own issue slots) is dedicated for consume instructions, which has also been adopted by prior research [3]. When any value arrives from the array, the corresponding consume instruction is executed, which broadcasts the value to the normal issue queue and wakes up dependent instructions.

State	Description
Modified (M)	line resides exclusively in this cache only content is modified relative to main memory
Exclusive (E)	line resides exclusively in this cache only content is same as main memory
Shared (S)	line resides in this cache but may be shared with other content is same as main memory
Invalid (I)	line contains no valid main memory copy

Table 4.1: In the MESI cache conherence protocol, each cache line is in one of four states: 1) modified, 2) exclusive, 3) shared, and 4) invalid.

Action	Current State	Next State	Bus Activity
Read	M	M	none (read hit)
Read	E	E	none (read hit)
Read	S	S	none (read hit)
Read	I	S/E	line fill (read miss) send inquiry to other caches
Write	M	M	none (write hit)
Write	E	M	none (write hit)
Write	S	S/E	write through (write hit) send invalidation signal to other caches
Write	I	I	write through (write miss) send invalidation signal to other caches

Table 4.2: State transfer of the cache line in the MESI cache conherence protocol when responding to messages initialized from the processor.

4.3.2 Supporting Multi-Trace Execution

In order to predict long traces accurately, Tracy speculatively dispatches multiple candidate traces at the same time. As long as one trace completes execution, the prediction is considered successful. Tracy uses the L1 data cache to hold the speculative program state produced by memory accesses, and each of its lines contains an extra bit to specify whether it has been speculatively modified. The cluster bus is segmented [108] accordingly to ensure that the copies of program state from different candidate traces can never be polluted by one another. Tracy also leverages and extends the underlying cache conherence protocol (MESI in this case) to support multi-trace execution. Note that this modification does not have any inherent dependencies on snooping-based protocols and can be ported easily to directory-based protocols.

In the MESI protocol, each cache line is in one of four states, which are described in Table 4.1. Table 4.2 and 4.3 list the state transfer of each cache line when responding to messages initialized from the processor and the snoopy bus, respectively. In order to support multi-trace execution, the MESI protocol is modified as follows.

- If any read or write miss occurs in the L1 data cache of a slave core (i.e., *slave cache*) and the

Action	Current State	Next State	Bus Activity
Read	M	S	provide data on bus and write back
Read	E	S	provide data on bus
Read	S	S	provide data on bus
Read	I	I	none
Write	M	I	provide data on bus and write back
Write	E	I	none
Write	S	I	none
Write	I	I	none

Table 4.3: State transfer of the cache line in the MESI cache conherence protocol when responding to messages initialized from the snoopy bus.

cache line cannot be provided locally by other *collaborative caches* that are occupied by the same candidate trace, the request is sent to the L1 data cache of the master core (i.e., *master cache*). If the cache line still cannot be provided, the L2 cache is then accessed. This extra process implicitly transfers live-in cache lines from sequential execution to the candidate trace. When receiving the write miss message, the master cache does not have to invalidate its own copy of the cache line because the candidate trace may abort later due to mis-speculation. On the contrary, when receiving the read miss message, the master cache does update the state of its cache line because it will remain valid in the requesting slave cache even when the corresponding candidate trace aborts.

- If any speculatively modified cache line is forced to be replaced in a slave cache, the candidate trace aborts. According to our experience, this situation rarely happens in practice.
- If a candidate trace aborts or is not selected to commit, all of its speculatively modified cache lines are invalidated. Otherwise, they are explicitly broadcasted to the master cache and all other slave caches as live-out cache lines. If the master cache cannot hold all these cache lines, they are written back to the L2 cache to become part of the permanent state. On the contrary, if any slave cache cannot hold all these cache lines, they are simply discarded. Finally, *exclusive* cache lines in all slave caches are changed to the *shared* state because they may have been requested by different candidate traces.

4.4 Summary

Tarcy is an innovative DBP technology that explores the possibility of leveraging hot traces to provide a large instruction window without introducing spurious dependencies. Inspired by multi-path execution [104, 105, 106], Tracy exploits the unique power of many-core architectures by launching

multiple traces and executing them simultaneously on idle cores. Multi-trace execution is essential to the capability of Tracy to exploit more coarse- and fine-grained parallelism than DIS- and CFG-based DBP techniques. For ILP, this technology enables the construction of very long traces that expose more distant parallelism opportunities and for LLP, this technology increases the possibility to break loop-carried dependencies that can not be handled by value prediction.

Multi-trace execution does require more cores and possibly more energy that can otherwise be used to execute other applications. In practice, however, this execution model is reasonable and efficient in many computing environments, such as high performance computing, research computing, cloud computing, and data centers. Furthermore, the CPU chip is only a small fraction of overall energy consumption. Large DRAM, disk, power supply inefficiencies, etc. are also major factors that impose a constant background “system leakage” while the machine is awake. “Wasting” some energy on the CPU chip to complete the task further and thus reduce energy spent on these other factors may be justifiable. For example, if the other factors are constant and the CPU chip consumes 30% of the total power, increasing the CPU chip power by 2x but reducing execution time by even a mere 1.3x is break-even on energy.

Tracy leverages and extends the underlying cache coherence protocol (MESI in this case) to support multi-trace execution. If any speculatively modified cache line is forced to be replaced in a slave cache, the candidate trace aborts. All cache lines that are speculatively modified by the committed trace are broadcasted to the master cache and all other slave caches as live-out cache lines. These cache lines become part of the permanent state and have to be written to the L2 cache if the master cache cannot hold all of them.

Chapter 5

Trace Construction and Prediction

High quality traces are the prerequisite for Tracy to effectively exploit both coarse- and fine-grained parallelism. More specifically, perfect traces have to simultaneously satisfy four requirements, which can be contrary to one another. First, traces have to be as long as possible to expose more distant parallelism opportunities. Second, traces have to be logically atomic. They should have a single entry point, a single exit point, and the control flow cannot exit prematurely through side exits. Thus, analysis can ignore all unnecessary control and data dependencies, enabling more aggressive parallelization. Third, traces have to be predicted accurately so that valuable CPU cycles and energy are not wasted on executing incorrect execution paths. Fourth, traces have to cover a large portion of dynamic instructions so as to produce large overall speedups.

5.1 Extending Branch Promotion

In prior research, rePlay [21, 46] constructs the longest atomic traces that can be predicted accurately. rePlay uses a bias table to keep track of whether each conditional branch has gone in the same direction for 32 consecutive times. If it has, the bias table indicates that the conditional branch should be promoted to an *assertion*, which is called *branch promotion*. Each instruction is appended to the end of the *trace construction buffer* when it retires. Once an unpromoted conditional branch is encountered, the pending trace is considered complete. A separate bias table is maintained for indirect branches and returns. For such control transfer instructions, a single bit for the last direction does not suffice and the target address must be kept in each entry. rePlay also associates each control transfer instruction with the global branch history [47]. This context sensitive information separates each control transfer instruction into instances based on the execution path leading up to itself.

Once separated this way, a greater number of control transfer instructions tend to exhibit biased behavior. The starting branch history of each trace (i.e., the branch history associated with the first control transfer instruction in the trace) is kept with the trace and only when the prior execution path matches this history, the corresponding trace is predicted to run.

rePlay is not compatible with multi-trace execution because it only constructs one trace under each different context. Thus, as the first step to further enlarge trace length, we relax the requirements of the original branch promotion heuristic so as to construct multiple longer traces under each different context, which can be predicted to run simultaneously. During program execution, each control transfer instruction is instrumented to track the frequency with which it jumps to each target address. As each instruction retires, it is appended to the end of the trace construction buffer, causing the pending trace to grow. Whenever a control transfer instruction is encountered and the frequency associated with its current target address exceeds 25%, it is promoted to be an assertion and treated as a normal instruction. Otherwise, the pending trace is terminated. An 11-element branch history is used to differentiate the same control transfer instruction under different contexts, assuming perfect alias resolution. Each trace is limited to be between 16 and 16K instructions long, and there are no restrictions on the number of traces that can be constructed. We use a simple trace prediction heuristic that dispatches every trace that 1) starts with the next target address, and 2) its starting branch history matches the prior execution path. If multiple candidate traces run to completion, the longest one is selected to commit.

We evaluate the above trace construction and prediction heuristics using the SPEC CPU2000 benchmark suite with the test data sets as input. Experimental results listed in Table 5.1 indicate that the constructed traces generally meet the four requirements to be considered high quality, especially for floating point applications. However, a more comprehensive analysis shows that some serious flaws exist by simply integrating branch promotion and multi-trace execution.

The third column of Table 5.1 lists the total size of unique traces that commit at least once during program execution. This is a conservative estimate of the working set size of traces, because a simple cache eviction policy [110, 111] could eliminate those traces that are constructed but never commit. Benchmarks like *perlbmk*, *swim*, *mgrid*, *applu*, *lucas*, *fma3d* and *apsi* do not experience serious code expansion through this technology. Because each trace can be always dispatched to the same core, the collaboration of all cores on the same chip can provide a large enough L1 instruction cache to hold the entire working set of traces. Furthermore, because traces only contain straight-line sequences of instructions, the traditional Next-N-Line prefetching scheme [112] works perfectly to fetch needed instructions into the L1 instruction cache on time. For half benchmarks, however, the

SPEC CPU2000 Benchmark		Trace Construction			Trace Prediction		Execution on Traces %
		Trace Size (MB)	Avg. Trace Length	Above 400 %	Accuracy	Avg. Candidate #	
INT	gzip	32.87	174.23	4.65%	96.61%	51.83, 5.41	99.49%
	vpr	20.06	111.24	1.31%	92.78%	66.31, 8.35	98.43%
	gcc	163.68	168.35	7.65%	85.18%	18.28, 4.10	94.07%
	mcf	17.34	112.47	1.44%	81.40%	32.58, 4.11	95.33%
	crafty	212.65	129.40	4.29%	91.51%	22.26, 3.19	95.92%
	parser	137.71	96.35	1.46%	88.50%	35.92, 6.40	96.70%
	eon	8.49	291.18	19.71%	88.94%	11.07, 3.00	90.37%
	perlbmk	1.29	318.23	16.78%	99.17%	5.20, 1.28	99.49%
	gap	45.08	153.37	4.79%	92.92%	16.92, 3.57	97.43%
	vortex	13.66	153.76	5.34%	99.11%	7.25, 1.60	99.85%
	bzip2	147.60	305.23	20.79%	98.53%	192.88, 14.85	99.75%
mcf	23.30	213.20	14.36%	86.34%	23.95, 4.99	92.15%	
FP	wupwise	20.91	555.08	28.73%	94.45%	4.61, 1.60	97.18%
	swim	1.42	984.85	10.14%	99.57%	7.08, 1.26	99.88%
	mgrid	3.95	7,548.63	98.60%	99.98%	8.03, 1.18	99.99%
	applu	2.62	6,892.11	95.96%	96.24%	2.63, 1.15	97.90%
	mesa	62.98	400.54	33.16%	99.71%	6.57, 2.60	99.60%
	galgel	35.22	801.19	80.33%	94.27%	33.23, 4.54	96.20%
	art	12.54	365.37	10.91%	98.65%	34.60, 2.30	99.84%
	equake	22.90	385.90	21.37%	83.33%	8.30, 1.75	89.88%
	facerec	13.89	2,745.72	79.88%	94.22%	11.25, 1.39	99.56%
	ammp	39.71	233.80	13.73%	94.28%	31.26, 2.44	99.06%
	lucas	2.97	363.19	1.83%	99.98%	52.04, 1.13	99.99%
	fma3d	0.57	416.87	27.83%	54.87%	6.67, 2.96	69.46%
	sixtrack	17.68	468.09	58.37%	96.42%	6.29, 1.95	99.10%
apsi	4.66	1,262.80	66.66%	99.90%	8.32, 1.56	99.97%	

Table 5.1: This table shows 1) the total size of unique traces that commit at least once, 2) the average length of traces that commit in each correct prediction, 3) the percentage of committed traces that are longer than 400 instructions, 4) the trace prediction accuracy, 5) the average size of the candidate trace pool and the average sorted rank of the committed trace in that pool, and 6) the percentage of instructions executed by the unmodified program that are covered by correctly predicted traces.

total size of useful traces is more than 20 MB, which is not likely to be even fit entirely in the L3 cache on modern microarchitectures. In most cases, a large portion of these traces are constructed within a small number of code regions that have complicated control flows, which can not be predicted accurately by only executing a small number of candidate traces. Thus, these code regions do not benefit from trace-based DBP and should continue executing sequentially.

The fourth column of Table 5.1 lists the average length (in instructions) of committed traces, weighted by the number of times that each trace commits. The fifth column lists the percentage of times that the committed trace contains more than 400 instructions. This threshold is selected due to our experience and such long traces have a higher probability to contain coarse-grained parallelism (e.g., LLP) that may produce larger speedups. For four floating point benchmarks (*mgrid*, *applu*, *facerec*, *apsi*), a majority of committed traces contain more than 400 instructions, with the average trace length in the thousands. Other floating point benchmarks and all integer benchmarks have shorter traces, in which fine-grained parallelism (e.g., ILP) is more likely to be exploited. However, after manually inspecting the constructed traces, we find that traces are terminated prematurely for most of the benchmarks. Recall that in the trace construction heuristic, only when the frequency associated with the current target address of the control transfer instruction exceeds 25%, it is promoted to an assertion and the pending trace continues growing. This requirement almost guarantees that the pending trace will be terminated after the innermost loop is exited. If there are not enough iterations in the innermost loop, the constructed trace is unnecessarily curtailed when more instructions should have been appended.

The first number in the seventh column of Table 5.1 is the average number of candidate traces per prediction, which forms the *candidate pool*. These numbers can be large, especially for integer benchmarks, which indicates that blindly executing all traces in the candidate pool would not be energy proportional to the performance increase and in many cases, the underlying architecture would not have that many cores to execute all of the candidate traces simultaneously. We then sort the candidate pool in decreasing order of the number of times each trace commits in prior program execution, with ties being broken by placing the longer traces first. The second number in the seventh column of Table 5.1 shows the average rank of the committed trace in the sorted candidate pool. Nine out of 14 floating point benchmarks have average rank of less than two and almost all benchmarks have average rank less than five. Thus, a simple heuristic to only execute a small number of high priority candidate traces in parallel is likely to substantially improve the energy proportionality while maintaining high prediction accuracy.

Compared to the original rePlay implementation, multi-trace execution dramatically increases

Parameter	Definition
MinTrLeng	minimum trace length
MaxTrLeng	maximum trace length
MinLoopLeng	minimum loop-derived trace length that is initially assigned
LoopDivisor	divisor to reduce minimum loop-derived trace length until minimum trace length is reached
MinBlkLeng	minimum length of a basic block to be a valid code structure
MaxTrNum	maximum number of traces that start with the same address
MinSpecAccuracy	minimum speculation accuracy for a code structure to remain valid when maximum trace number is reached
TrDiscThold	number of times that traces derived from a code structure are allowed be discarded consecutively
TrConstThold	number of times that a trace has to repeat before being officially constructed

Table 5.2: Parameter definition of the trace construction algorithm.

trace length while achieving higher speculation accuracy and larger coverage of dynamic instructions. Simply integrating branch promotion and multi-trace execution, however, has two drawbacks. On one hand, branch promotion sometimes overuses multi-trace execution by constructing traces from code regions that have complicated control flows, which do not benefit from trace-based DBP and should continue executing sequentially. On the other hand, branch promotion sometimes underuses multi-trace execution by terminating traces prematurely when the innermost loop is exited. Thus, it is both important and necessary to develop novel trace construction algorithms that fully but carefully exploit the unique power of multi-trace execution.

5.2 Exploiting Hierarchical Code Structures

Based on the above observations, We develop an innovatative trace construction algorithm that holistically balances among trace length, speculation accuracy, and coverage of dynamic instructions. Tracy constructs the longest traces that can be accurately speculated on the available number of cores. In certain code regions that have complicated control flows, Tracy stops constructing traces and executes these code regions sequentially.

In order to reduce runtime overhead, Tracy implements dynamic instrumentation in hardware by directly integrating trace construction with instruction pipeline execution, which has been widely adopted by prior research [18, 20, 21, 25, 40]. As illustrated in Figure 5.1, Tracy constructs traces at the retire stage of each instruction to avoid being on the critical path of pipeline execution. It stores real traces in main memory and their metadata in the set associative *trace cache* on chip. Each trace is indexed by its start address and placed in the corresponding set. These metadata are used

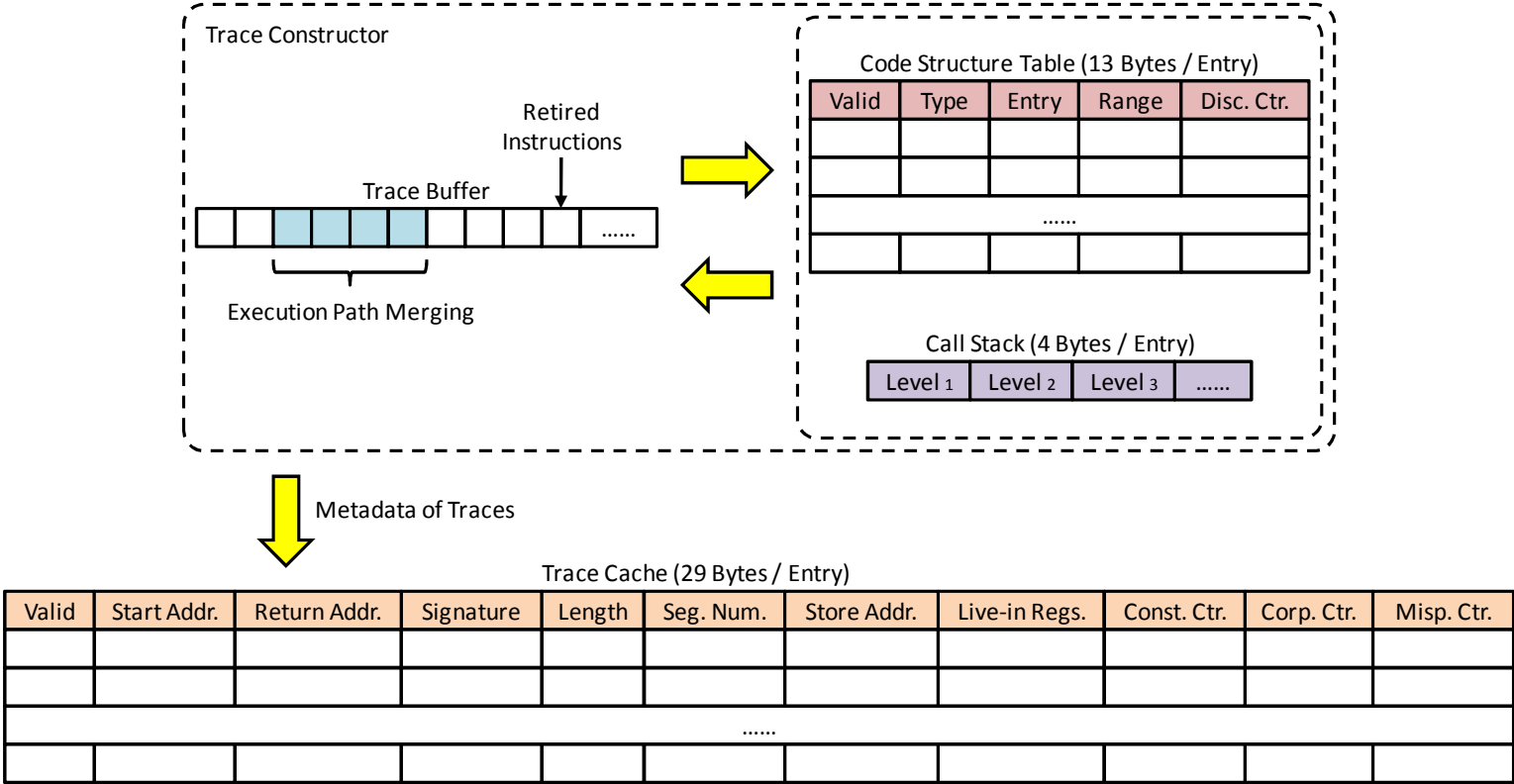


Figure 5.1: Tracy constructs traces at the retire stage of each instruction. It stores traces in main memory and their metadata in the set associative trace cache on chip.

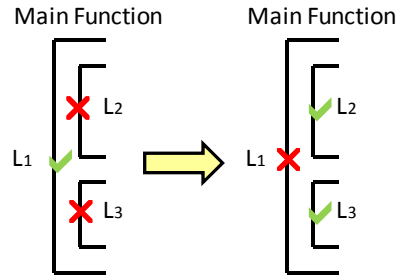


Figure 5.2: Tracy starts to exploit code structures in the outermost scope, and only enters the next level if necessary.

for different purposes and will be explained through the entire section. Tracy constructs traces in three steps, each of which will be described in detail in the following sections. All parameters used in the algorithm are defined in Figure 5.2.

5.2.1 Selecting Starting and Ending Points

When a program starts running, Tracy launches a thread on a separate core to analyze the binary executable and build the hierarchy of *code structures* [113], which are stored in main memory. Tracy uses these code structures to define the starting and ending points of each *primitive trace*, which may contain 1) one complete function invocation, 2) one or several complete iterations of the same loop, or 3) one basic block of more than *MinBlkLeng* instructions. In order to maximize their length, traces are initially restricted to start and end at the outermost functions or loops, whose *entries* and *ranges* are loaded into the *code structure table (CST)* from main memory. During program execution, if a code structure shows unpredictable internal execution paths, it is deleted from the CST and the next level of inner code structures is loaded. For example, Figure 5.2 illustrates a simple program comprising an outer loop L_1 and two separate inner loops L_2 and L_3 . Initially, traces can only start with the head of L_1 . If these traces can not be accurately predicted, the head of L_1 is no longer used to initialize traces and the heads of L_2 and L_3 are used instead. These inner loops produce a smaller number of shorter execution paths so that correct traces starting from the corresponding program points will be easier to predict.

Tracy ends a primitive trace under one of two possible conditions: 1) the target of its last instruction goes beyond the range of the corresponding code structure, and 2) it is derived from a loop, the target of its last instruction goes back to the loop entry, and it is more than *MinLoopLeng* instructions long. The second condition is used to partition the iteration space in case the entire loop execution is too large to fit into a single primitive trace. If the first primitive trace is already

more than *MinTrLeng* instructions long, the entire trace terminates. Otherwise, Tracy continues appending instructions until the next found primitive trace ends. This process may repeat multiple times until the *compound trace* finally reaches the minimum trace length. In order to construct longer traces, *MinLoopLeng* is initially assigned a much larger value than *MinTrLeng*. Because both functions and loops can include subroutine invocations, Tracy also maintains a *call stack* for the primitive trace under construction. Only when the call stack becomes empty, the control flow is in the correct core structure so that the end condition testing is necessary.

5.2.2 Appending Retired Instructions

Once the starting and ending points are selected, Tracy indexes the address of each retired instruction in the CST, and begins a new trace if it matches the entry of any code structure. After that, every retired instruction is appended to the end of the *trace buffer*, with all control transfer instructions (e.g., conditional branches, indirect branches, returns) changed to assertions [21, 46]. An assertion encodes the pre-determined target of the original control transfer instruction based on the particular trace, and sends an abort signal to Tracy if it deviates from the execution path that is actually taken. If an instruction invokes a system call whose effects may not be recoverable, or the maximum trace length of *MaxTrLeng* instructions has already been reached, the trace under construction is discarded. Each CST entry contains a field (*Disc. Ctr.*) to record the number of consecutive times that a trace derived from this code structure is discarded.

5.2.3 Inserting into the Trace Cache

Four pieces of metadata in each trace cache entry uniquely determine a trace, including 1) the address of its first instruction (*Start Addr.*), 2) the address of the next instruction to be executed if it runs to completion (*Return Addr.*), 3) the signature by XORing all its instructions (*Signature*), and 4) the total number of its instructions (*Leng*). All addresses are based on the original program. Another piece of metadata is also maintained to count the number of times that the trace has repeated (*Const. Ctr.*). If the newly terminated trace is found in the trace cache and has already repeated *TrConstThold* times, it is *officially* constructed and the trace buffer is passed to the trace parallelizer for further processing. If the trace has never been encountered before, Tracy then counts the number of existing traces that start with the same address. If the value has already reached *MaxTrNum*, the new trace can only replace an old trace with the same start address. Otherwise, it can be inserted into an empty entry if available.

A code structure is considered inappropriate to construct traces when one of two conditions are encountered. First, the derived traces have been discarded *TrDiscThold* times consecutively. Second, the number of derived traces has reached *MaxTrNum* but the corresponding speculation accuracy is below *MinSpecAccuracy*. Both conditions indicate that trace length needs to be reduced. If the code structure is a loop and the value of *MinLoopLeng* divided by the constant *LoopDivisor* is still larger or equal to the value of *MinTrLeng*, the code structure remains in the CST and *MinLoopLeng* is updated with the newly calculated value. Otherwise, the code structure is deleted from the CST and the next level of inner code structures is loaded. In both cases, all existing traces derived from the code structure are deleted from the trace cache. One exception in the second case is that when any compound trace exists, the code structure also gets another chance to remain in the CST. The compound trace is, however, deleted from the trace cache and the code structure that starts the last primitive trace is no longer allowed to extend any traces that are initially derived from the code structure that starts the first primitive trace.

5.3 Adaptive Speculation

Trace prediction occurs simultaneously with normal instruction fetch and introduces negligible runtime overhead [18, 20, 21, 25, 40]. Each trace cache entry contains two fields that are correlated to trace prediction. The *Corp. Ctr.* field counts the number of times that a trace has been correctly predicted. For traces that start with the same address, if one counter reaches the maximum value, the values in all counters are halved in order to maintain the correct ranking of traces based on their hotness. On the other hand, the *Misp. Ctr.* field counts the number of times that a trace has been mispredicted consecutively. When the counter value exceeds the threshold (defined as *TrDisbThold*), the trace is disabled temporarily and can only be enabled again after being identified several more times (defined as *TrDisbPenalty*). Tracy launches as many enabled traces as it can that start with the next executed instruction. Priority is given to traces that are correctly predicted more frequently, with ties being broken by selecting the longer traces.

During dispatching, each trace is assigned to a core that it has been executed on before, which is maintained by Tracy. If it is not possible due to assignment conflicts or this is the first time that the trace is speculated, Tracy dispatches it to the core that holds the least size of traces. This design is to evenly distribute traces to different L1 instruction caches.

Each trace cache entry contains three fields that are necessary to execute the candidate trace: 1) the *Live-in Regs* field lists the live-in registers of the trace, 2) the *Seg. Num.* field designates the

number of parallelized threads, and 3) the *Store Addr.* field contains the real start address of the trace in main memory. Live-in registers are transferred in bulk from sequential execution to all the candidate traces. If multiple traces run to completion, program state from the longest one is selected to commit, whose live-out registers are transferred back to sequential execution. Information of live-out registers is patched at the end of each parallelized thread after parallelization is performed, so that it does not need to be maintained in the trace cache. The transfer of live-in and -out cache lines has been described in Section 4.3.2.

5.4 Experimental Setup

Tracy is implemented by heavily extending the SESC simulation framework [114]. In the following subsections, we will describe in detail how the architectures, algorithms, benchmarks, and evaluation methodology are selected and configured in our experiments.

5.4.1 Architectures

We evaluate Tracy on a symmetric many-core architecture with 34 2-issue IO cores. A single master cluster contains two cores, one for trace management plus sequential execution, and the other one for trace parallelization. The remaining 32 cores are equally divided into four slave clusters to execute the parallelized candidate traces.

The design of putting a number of identical single-issue or dual-issue IO cores onto a single chip has been adopted by many classic products, such as the Raw processor [33], the Tiler processor [115], the UltraSPARC T1/T2 processor [116], the Larrabee processor [117], and the OCTEON processor [118]. More recently, Intel announces its MIC architecture [119], which follows the same design direction. Knights Corner, the first commercial MIC architecture product, will be manufactured using Intel’s latest 3D Tri-Gate 22nm transistor process and will feature more than 50 IO cores. Thus, we believe that integrating a number of IO cores onto a single chip will continue to be one of the major many-core architecture designs in the future.

The entire system can be configured by varying three independent parameters: 1) the number of cores to use, 2) the number of parallel threads that each trace is decomposed into, and 3) the core type. For example, if the architecture is composed of 2-issue IO cores and Tracy uses all 34 cores to support 4-way parallelization for each trace, the configuration is represented as $\text{Tracy}_{34}^4\text{-io2}$. Similarly, the corresponding single-threaded execution on this architecture is represented as ST-io2 . Table 5.3 shows the architectural parameters of $\text{Tracy}_{34}^4\text{-io2}$, our major experimental system. The

Substrate	2-issue IO Core	2.0 GHz, 2/2/2 fetch/issue/retire width, 1/1/1/1 INT ALU/FP ALU/AGU/Branch Unit, 1/1 INT Multiplier/FP Multiplier, 1/1 INT Divider/FP Divider, 32+40/32+40 INT/FP registers, Hybrid Branch Predictor, 11-bit history, 16 KB Global/Local/Meta Table
	L1 Instruction Cache	32 KB, 4-way associative, 2 cycles access latency, 16 MSHRs, WT, LRU
	L1 Data Cache	64 KB, 4-way associative, 2 cycles access latency, 32 MSHRs, WT, LRU
	Shared L2 Cache	8 MB, 16-way associative, 10 cycles access latency, 64 MSHRs, WB, LRU
	Cluster Bus	256 bit width for cache access, 1 cycle delay
	Backbone Bus	256 bit width, 3 cycles delay (1 cycle per segment)
	Memory Bus	64 bit width, 15 cycles delay
	Memory	400 cycles access latency
Tracy	Code Structure Table	4K function entries, 4K loop entries, 4K basic block entries, 16-way associative
	Trace Buffer	16K instructions
	Trace Cache	16K entries, 64-way associative
	Synchronization Array	24K entries, 1 cycle access latency

Table 5.3: Architectural parameters of Tracy₃₄⁴-io2.

MinTrLeng	64
MaxTrLeng	16,384
MinLoopLeng	4,096
LoopDivisor	4
MinBlkLeng	8
MaxTrNum	32
MinSpecAccuracy	93.75%
TrDiscThold	4
TrConstThold	16
TrDisbThold	32
TrDisbPenalty	1

Table 5.4: Algorithmic parameters of trace construction and prediction.

synchronization array is divided into two banks, each having 12K entries and four requesting ports. For each core, the dedicated link to the array can support two produce and two consume messages per clock cycle. Note that Tracy is also effective even with much smaller core counts or much more advanced core types. In order to comprehensively evaluate the performance of Tracy, we will also conduct various sensitivity analyses using different system configurations.

5.4.2 Algorithms

Figure 5.4 shows the parameters of the trace construction and prediction algorithms used by Tracy. These parameters are selected because a combination of them achieves good performance on average

in our design space exploration of these algorithms.

5.4.3 Benchmarks

We evaluate Tracy on the SPEC CPU2000 (both integer and floating point) and MediaBench benchmark suites. This choice represents a wide range of programs in terms of parallelization difficulty, where integer applications are the hardest to parallelize and floating point applications are the easiest. For SPEC CPU2000, we select the test data sets as input, and for MediaBench, we use real world images, audios, and videos downloaded from the Internet. All benchmarks are compiled using GCC 4.4 -O2 to include the effect of static optimization.

5.4.4 Evaluation Methodology

As described in Section 5.2.3, each constructed trace is passed to the trace parallelizer for further processing. Thus, there is a lag between the time that the trace is constructed and the time that the trace can be actually executed to provide parallel performance. Although this parallelization time can be amortized when the program runs long enough, it becomes significant if we want to maintain a reasonable simulation time. If we do not count parallelization time, however, some long traces with short lifetime may result in overstated performance. Thus, we execute each benchmark two times. In the first time, the program runs under functional emulation mode and trace parallelization time is configured to be zero. In the second time, the program runs under cycle-accurate simulation mode (at most one billion instructions) by only using the traces that remain in the trace cache at the end of the first run.

Prior research [14, 15, 18, 20, 21], including our own experimental results have demonstrated that programs tend to repeat long sequences of instructions (i.e., traces). For each program phase, a small set of traces usually covers most of dynamic instructions. As program phases repeat during execution, these traces get executed continuously while their parallelization time only needs to be charged once. After all program phases have been encountered enough times so that all traces have been parallelized, the program enters the steady state and at that time, its performance is considered similar to the performance simulated in our experiments. Similar evaluation methodologies have also been adopted by prior research [6, 7, 13, 21] in the DBO and DBP area to evaluate the performance of a program in its steady state.

Two situations may cause the parallelization time to be charged multiple times. First, some code regions may contain complicated control flows so that the same trace is first constructed, then

gets deleted from the trace cache, and later is constructed again. However, Tracy stops constructing traces in these code regions and executes them sequentially. These effects are accurately captured by our experiments. Second, the same code region may produce several different program phases that are dominated by different set of traces. Because these set of traces probably share many start addresses, they may frequently evict one another from the trace cache if these program phases are tightly interleaved. However, Tracy can maintain the parallelized version of the trace in main memory even when it is deleted from the on-chip hardware trace cache. Thus, when the same trace is constructed again due to program phase change, the parallelization process is actually turned into hash table searching, whose runtime overhead is negligible. Furthermore, traces that are constructed from prior runs of the program can be shipped with the binary executable and loaded into the trace cache at the start of any future runs. If these traces do not match the execution paths which are actually taken, they will be quickly evicted from the trace cache and new traces will be constructed instead. However, if the program shows similar behaviors with those of prior runs, these traces can be executed immediately because they are already parallelized. This technology has a high probability to make short-running programs also benefit from trace-based DBP.

The prototype implementation of Tracy actually causes large overhead to process traces. For the longest trace, Tracy may need several seconds for optimization and parallelization and even several minutes for register allocation. This performance is far worse than the performance of state-of-the-art compilers, such as GCC and LLVM. Thus, we believe the implementation inefficiency can be largely eliminated by carefully tuning the current algorithms.

5.5 Experimental Results

Table 5.5 contains the statistical analysis of the trace construction and prediction algorithms for each benchmark. The system is configured as Tracy₃₄⁴-io2 so that at most eight candidate traces can run simultaneously. Column 3 contains the percentage of instructions executed by the unmodified program that are contained in correctly predicted traces. This number is typically over 90% for floating point benchmarks, which indicates that almost all dynamic instructions can be executed in parallel. However, the average percentage for media and integer benchmarks is only 62.04% and 47.74%, respectively, leaving a large portion of the program to run sequentially. Column 7 indicates that the average trace length varies drastically for each benchmark, ranging from 100s of instructions for integer benchmarks to 1000s of instructions for floating point benchmarks. Longer traces expose more parallelism opportunities and typically lead to better parallel performance.

Benchmark	Exec. on Traces %	Trace #	Misp. Rate	Avg. Candidate #	Avg. Trace Length	
FP	wupwise	99.36 %	82	0.01 %	1.43	4,196
	swim	95.68 %	129	7.96 %	2.55	521
	mgrid	99.92 %	192	0.20 %	1.15	8,291
	applu	99.35 %	95	2.60 %	1.84	4,251
	mesa	99.52 %	58	0.37 %	2.57	1,497
	equake	94.55 %	141	3.38 %	4.03	600
	ammp	80.15 %	435	2.75 %	3.01	287
	sixtrack	92.70 %	1,191	0.97 %	2.42	100
MEDIA	epic-dec	88.41 %	93	3.28 %	2.43	91
	epic-enc	93.22 %	47	13.40 %	1.98	1,172
	g721-dec	64.94 %	135	9.19 %	4.76	81
	g721-enc	48.15 %	77	8.53 %	4.10	75
	gsm-dec	97.84 %	84	12.93 %	5.38	1,010
	gsm-enc	94.90 %	222	5.18 %	4.56	519
	jpeg-dec	77.18 %	203	3.31 %	2.77	562
	jpeg-enc	35.80 %	171	9.12 %	2.47	357
	mpeg2-dec	86.87 %	300	6.65 %	2.86	176
	mpeg2-enc	14.71 %	354	13.50 %	3.95	193
INT	gzip	50.35 %	493	13.03 %	3.05	92
	crafty	57.69 %	3,778	2.54 %	3.33	72
	parser	33.57 %	1,390	4.42 %	3.07	99
	eon	74.94 %	437	2.58 %	3.60	104
	bzip2	33.94 %	479	3.38 %	2.85	114

Table 5.5: This table shows trace-related statistical analysis of Tracy₃₄⁴-io2, including 1) the percentage of instructions executed by the unmodified program that are covered by correctly predicted traces, 2) the number of traces in the trace cache, 3) the trace misprediction rate, 4) the average number of candidate traces for each prediction, and 5) the average length of the trace that commits in each correct prediction.

Benchmark		Exec. on Traces %	Trace #	Misp. Rate	Avg. Candidate #	Avg. Trace Length
FP	Tracy ₁₀ ⁴ -io2	91.98 %	186	2.20 %	1.56	1,063
	Tracy ₁₈ ⁴ -io2	93.91 %	172	1.17 %	2.03	1,092
	Tracy ₃₄ ⁴ -io2	94.93 %	173	0.78 %	2.22	1,090
MEDIA	Tracy ₁₀ ⁴ -io2	55.75 %	138	17.65 %	1.74	282
	Tracy ₁₈ ⁴ -io2	59.81 %	135	9.51 %	2.55	256
	Tracy ₃₄ ⁴ -io2	62.04 %	141	7.58 %	3.35	273
INT	Tracy ₁₀ ⁴ -io2	40.89 %	878	17.96 %	1.80	94
	Tracy ₁₈ ⁴ -io2	46.62 %	877	7.46 %	2.64	95
	Tracy ₃₄ ⁴ -io2	47.74 %	885	4.18 %	3.17	95

Table 5.6: This table summarizes trace-related statistical analysis of 1) Tracy₁₀⁴-io2, 2) Tracy₁₈⁴-io2, and 3) Tracy₃₄⁴-io2, including 1) the percentage of instructions executed by the unmodified program that are covered by correctly predicted traces, 2) the number of traces in the trace cache, 3) the trace misprediction rate, 4) the average number of candidate traces for each prediction, and 5) the average length of the trace that commits in each correct prediction.

Column 6 illustrates that for most benchmarks, only five or fewer candidate traces are actually dispatched on average, despite the possibility to execute up to eight traces on all available cores. In practice, far fewer than five candidate traces actually run at any given time, because failed speculations end more quickly on average while successful speculations run to completion. At the same time, Column 5 shows that the fraction of dispatches for which all traces abort is less than 7% in two-thirds of all benchmarks. Our trace construction algorithm sacrifices trace length and dynamic execution coverage if necessary in order to achieve high speculation accuracy, so that valuable CPU time and energy are not wasted on executing incorrect execution paths. Finally, Column 4 shows the number of traces that are left in the trace cache at the end of the first run. Even though Tracy may have constructed many more traces, most of them are deleted afterward and only a small number of traces actually reside in the trace cache when the program enters the steady state of execution. Except for three benchmarks (*sixtrack*, *crafty*, *parser*), all other applications generate fewer than 500 traces, and in more than half cases, the number is less than 200.

In summary, for all floating point benchmarks and more than half media benchmarks, the constructed traces are considered to have high quality by simultaneously satisfying four requirements. For example, *mgrid* has an average trace length of 8,291 instructions together with 99.80% speculation accuracy and 99.92% coverage of dynamic instructions. Other media benchmarks and all integer benchmarks, however, have relatively unpredictable control flows, leading to shorter trace length and lower dynamic execution coverage. In the limit study, Figure 3.5(a) shows that the average trace length for integer benchmarks is 235 basic blocks and Figure 3.6(a) shows that only 0.02% basic blocks are not formed into traces on average. This large performance gap is mainly caused by the assumption of an unbounded number of cores in the limit study, in which speculation is always successful as long as the correct trace has been constructed. In the real world, however, Tracy has to shorten traces or even give up parallelizing certain code regions if the available number of cores cannot execute all candidate traces simultaneously.

Table 5.6 summarizes the performance of the trace construction and prediction algorithms when the system is configured as 1) Tracy₁₀⁴-io2 (at most two candidate traces), 2) Tracy₁₈⁴-io2 (at most four candidate traces), and 3) Tracy₃₄⁴-io2 (at most eight candidate traces), respectively. For floating point benchmarks, decreasing the number of maximum candidate traces has limited impact on every evaluated metric. In this case, even when cores are available to execute many candidate traces simultaneously, it may be more economical for Tracy to only use part of them, which are just enough to achieve near-optimal performance with much less energy consumption. For media and integer benchmarks, however, the performance of Tracy is more sensitive to the number of available

cores. While four candidate traces are still enough to maintain high speculation accuracy in most cases, the availability of only two candidate traces dramatically increases the misprediction rate to 17.65% for media benchmarks and 17.96% for integer benchmarks. In this case, Tracy has to be provided with enough cores in order to achieve reasonable speedups.

5.6 Summary

In this chapter, we present an innovative trace construction algorithm that holistically balances among trace length, speculation accuracy, and coverage of dynamic instructions. Tracy constructs the longest traces that can be accurately speculated on the available number of cores. In certain code regions that have complicated control flows, Tracy stops constructing traces and executes these code regions sequentially. Tracy exploits the unique power of many-core architectures by launching multiple traces and executing them simultaneously on idle cores. The major insight is that in many cases, speculation accuracy can be *dramatically* increased by only trying a *very small* number of candidate traces. Simply integrating multi-trace execution with branch promotion, the state-of-the-art trace construction strategy, has two drawbacks. On one hand, branch promotion sometimes overuses multi-trace execution by constructing traces from code regions that have complicated control flows, which do not benefit from trace-based DBP and should continue executing sequentially. On the other hand, branch promotion sometimes underuses multi-trace execution by terminating traces prematurely when the innermost loop is exited.

When eight candidate traces can be executed simultaneously, high quality traces can be generated for all floating point benchmarks and more than half media benchmarks. Averaged over all floating point benchmarks, for example, Tracy constructs traces of more than one thousand instructions and simultaneously achieves speculation accuracy of 99% by only executing about two traces at the same time. Furthermore, 95% of the instructions executed by the unmodified program are covered by correctly predicted traces. Thus, a solid foundation is set for Tracy to achieve great speedups by optimizing and parallelizing the constructed traces. When the number of maximum candidate traces is decreased to two, however, the speculation accuracy is dramatically decreased for media (from 7.58% to 17.65%) and integer (from 4.18% to 17.96%) benchmarks, suggesting that the number of available cores is not enough to operate Tracy effectively.

Chapter 6

Trace Optimization

As illustrated in Figure 6.1, Tracy optimizes and parallelizes the constructed trace in five steps. First, the single static assignment (SSA) form is constructed to eliminate anti and output data dependencies among registers. Second, various optimizations are performed to eliminate unnecessary data dependencies (both register and memory) or increase data dependency lengths, exposing more parallelism opportunities. After that, the dependency graph is constructed based on the optimized trace, on which various parallelization strategies are then performed to partition and schedule instructions across multiple cores and insert necessary synchronizations to maintain the correct register and memory access order. Finally, a graph-based register allocation algorithm [120, 121] is performed to each parallelized thread separately.

For each core, we dedicate a software-managed scratchpad memory to be the spill space [122, 123, 124, 125], because these memory references are always private (i.e., not affected by cache coherence as well as multi-trace execution) and the values are simply discarded after the trace runs to completion or aborts. The spill space is connected to the processor directly so that register spilling inserted by Tracy does not interfere with normal data access. Furthermore, the dedicated space prevents spilled registers from being polluted by other data. In the current design, Tracy ensures that the number of spilled registers is less than the number of entries in the spill space so that every entry is used to spill at most one register. Because each entry is written at most once during trace execution, store buffering is no longer needed. This property is especially useful in OoO execution because the spill instructions only have RAW dependencies and do not need to compete for the load/store queue with normal memory loads and stores. Instead, a simple lookup table is maintained, which contains one bit for each entry in the spill space. Every spill write instruction sets the corresponding bit

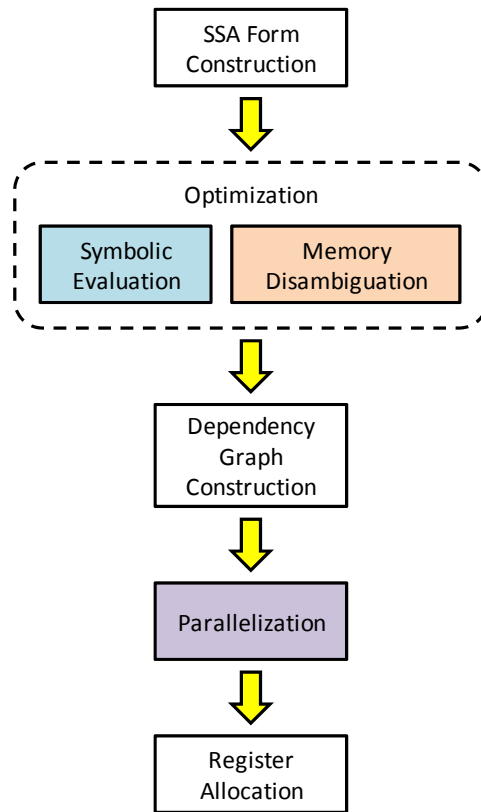


Figure 6.1: Tracy optimizes and parallelizes the constructed trace in five steps.

after it has been executed and broadcasts the value to be stored to the dedicated issue queue (with its own issue slots) for spill read instructions (similar to load-store forwarding), which only enter the execution stage if the corresponding bit is set. The value to be stored is then broadcasted to the normal issue queue to wake up dependent instructions. The same architectural optimizations applied to produce/consume instructions are analogously applied to spill read/write instructions. First, if an instruction needs to read multiple values from the spill space, they are encoded into one variable-length instruction. Second, the entry number of the spill space is simply extracted after the spill write instruction has been decoded and is directly used by the instruction that actually produces the value to perform spill writing.

The spill space described above is specially designed for atomic traces and cannot be generalized to facilitate traditional program execution. Because any pair of functions has the possibility to be in the same call stack, the entire space has to be divided among different functions with all parts being mutually exclusive. This solution, however, still does not work for recursive function calls. Thus, traditional compilers spill registers on the stack and the memory address must be calculated using the stack pointer during run time.

This chapter describes two major optimizations performed by Tracy, which are symbolic evaluation and memory disambiguation. The functionality of these optimizations is not only to directly produce speedups, but more importantly, to prepare the code for future parallelization. Thus, the performed code transformations may be suboptimal for increasing the program performance by themselves, but they reformat the code to be more amenable to parallelism. Furthermore, these optimizations are designed to fully exploit the atomicity property of traces, within the confines of the underlying architectural support.

6.1 Symbolic Execution

Like compilers, CFG-based DBP techniques typically rely on various data-flow analyses [29] to optimize programs. They set up data-flow equations for each node of the CFG and solve them by repeatedly calculating the output from the input locally at each node until the entire system stabilizes, i.e., it reaches a fixpoint. In order to handle the large number of execution paths represented by the CFG, diverged program states are conservatively merged at certain joint points. Furthermore, the name space of data-flow functions is typically based on lexical names of variables, leaving many optimization opportunities behind. Several DBO systems [21, 32] have also constructed atomic traces, but they simply adopt some traditional optimization algorithms and do not customize them to leverage the unique opportunity. Because each atomic trace only represents a single execution path with all control dependencies and derived data dependencies being ignored, analysis scalability is no longer a problem. Thus, it is both important and necessary to design heavyweight but powerful optimization algorithms to fully exploit the atomicity property of traces.

Tracy, on the contrary, leverages symbolic evaluation [31] to aggressively optimize traces. This technology assigns a symbolic value to each defined register or memory location (abbreviated as *loc*). Code analysis and transformations are then performed through symbolically executing each program path and updating these values. The path sensitivity characteristic of symbolic evaluation is not scalable, and thus has not been used by the compiler that needs to optimize the entire CFG. However, because symbolic evaluation performs path-sensitive program analysis and data-flow information is based on symbolic values instead of lexical names, it performs more precise program analysis than traditional data-flow analysis [29]. While symbolic evaluation has also been adopted by prior research [28] to perform pointer analysis on hyperblocks, we extend it to perform general optimizations on atomic traces.

Tracy divides all MIPS (ISA supported by the SESC simulation framework) instructions into four categories: 1) memory loads, 2) memory stores, 3) instructions that only perform integer arithmetic and logical operations (including control transfer instructions that test integer registers), and 4) all other instructions. Each symbolic value is represented as $C_0 * V_0 + C_1 * V_1 + \dots + C_{n-1} * V_{n-1} + C_n$, in which each C_i is a constant and each V_i is another symbolic value. The last C_n is called the *immediate part* of the symbolic value. Thus, Tracy does not model any multiplications and divisions, which require special registers to be updated. Furthermore, Tracy does not model any calculations involving floating point registers, which are not associative due to conversion and rounding. Although some solutions have been proposed in the area of model checking [126, 127] and formal verification [128, 129] to support non-linear arithmetic constraints and floating point operations, they have not been used in program optimization.

Figure 6.2(a) illustrates the algorithm to symbolically evaluate memory loads. First of all, the effective address is calculated (line 1) and the memory value is loaded from the corresponding location (line 2). We then search for the *earliest defined* live register, whose value is either *same* to the loaded value or *only* differs from it in the immediate part. If such *substitute register* exists, this memory load is redundant and we simply replace it by adding the immediate difference to the substitute register (line 6). Otherwise, the memory load cannot be eliminated and we instead search for the substitute to the base register itself and replace it with the one that is found (line 8). Finally, the destination register is updated with the loaded value (line 11). Two major operations need to be processed when symbolically evaluating memory stores, whose algorithm is illustrated in Figure 6.2(b). First, if the pending value is the same as the value that is already stored in the corresponding location, the memory store is redundant and can be simply eliminated (line 6). Second, if the memory store cannot be eliminated, the stored value in all potentially aliased locations needs to be updated (line 9). In Figure 6.2(c), we show how to symbolically evaluate the add operation, which is one example of instructions that only perform integer arithmetic and logical operations. Other instructions in this category are processed with similar but customized algorithms. Similarly, we first search for the substitute register of the added value (line 4) and if it does not exist, we instead search for the substitute to each of the source register (lines 9 to 10). In the latter case, however, the substitute register must hold the exactly same value as the original register because there is no immediate field in the particular instruction. Figure 6.2(d) illustrates the algorithm to symbolically evaluate all other instructions that include non-linear or floating point calculations, in which only attempted substitutions of source registers are performed (lines 1 to 2). After all instructions in the trace have been symbolically evaluated, we then perform *register coalescing* to eliminate unnecessary

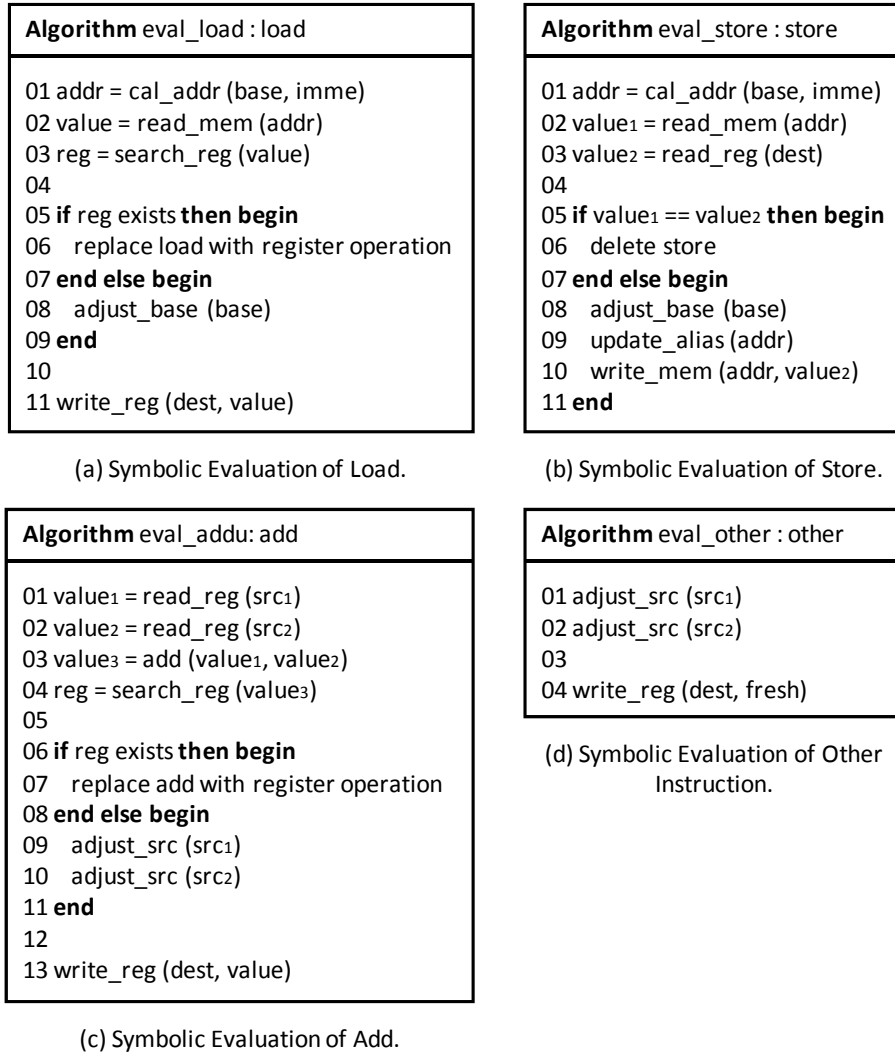
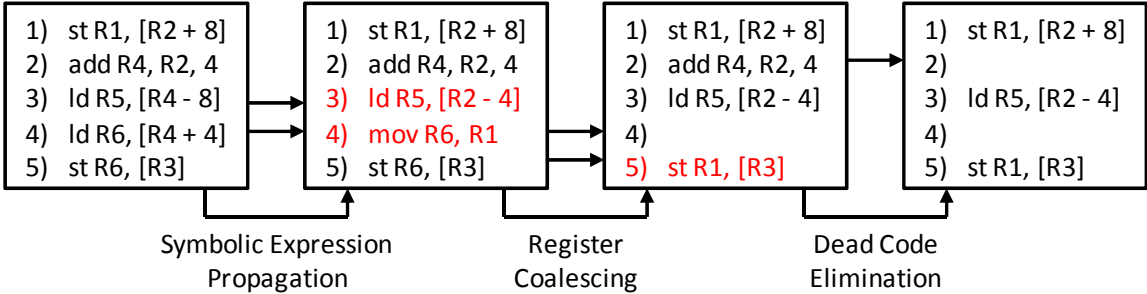


Figure 6.2: Tracy divides all MIPS instructions into four categories: 1) memory loads, 2) memory stores, 3) instructions that only perform integer arithmetic and logical operations (including control transfer instructions that test integer registers), and 4) all other instructions. Tracy uses different strategies to symbolically evaluate instructions in different categories.

register copies. After that, we perform the final round of *dead code elimination* to remove instructions whose destination register or memory values are no longer used in future program execution, except those that need to live out of the trace.

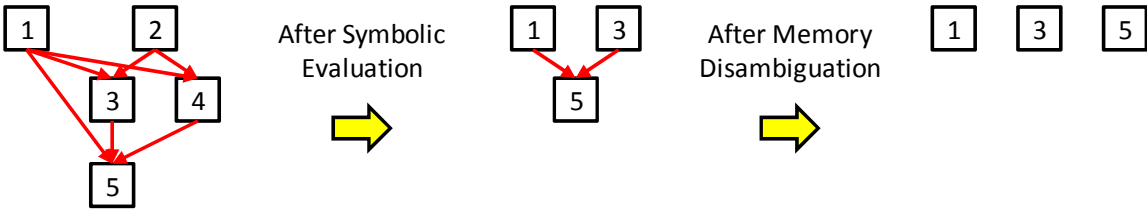
The fundamental principle of the above optimizations is to reuse as many registers as possible that are defined early in the trace, after which, unnecessary and dead register copies can be simply eliminated to increase program performance. Tracking symbolic values instead of lexical names on the single execution path, symbolic evaluation combines the effects of various traditional optimizations, including constant propagation, value propagation, value numbering, common subexpression elimination, loop unrolling, loop-invariant code motion, redundancy elimination, and dead code elim-



(a) Symbolic Evaluation.

Group1 (instructions 1 and 3): base register = R2, address range = [R2 - 4, R2 + 12)
 Group2 (instruction 5): base register = R3, address range = [R3, R3 + 4)
 Non-Alias Condition Test: $(R2 - 4 \geq R3 + 4) \ || \ (R2 + 12 \leq R3)$

(b) Memory Disambiguation.



(c) Dependency Graphs.

Figure 6.3: Tracy performs symbolic evaluation and memory disambiguation before parallelization to eliminate unnecessary data dependencies (both register and memory) or increase data dependency lengths, exposing more parallelism opportunities..

ination. These optimizations not only produce speedups by themselves, but more importantly, they also eliminate unnecessary data dependencies (both register and memory) or increase data dependency lengths, exposing more parallelism opportunities. In an extreme case, if all instructions in the trace can be transformed to only use live-in registers, they can be partitioned and scheduled without any constraints to fully exploit the processing power of the underlying many-core architecture. Finally, the optimizations performed on memory loads and stores always replace the base register with an earlier defined register if possible. As a consequence, a larger number of memory loads and stores share the same base register. As will be described in Section 6.2, this code transformation is essential to the memory disambiguation algorithm used by Tracy.

Figure 6.3(a) illustrates a small trace snippet with five instructions. Initially, the live-in registers R_1 and R_2 are assigned with symbolic values V_1 and V_2 . After the first two instructions are executed, V_1 is stored in $loc[V_2+8]$ and R_4 is assigned with V_2+4 after symbolically executing the add operation. At instruction 3, because $R_4 - 8$ is evaluated to be $V_2 - 4$, the load address can be transformed to $R_2 - 4$, which is only dependent on the live-in registers. Similarly, instruction 4 loads from $loc[V_2+8]$, which happens to hold the symbolic value V_1 stored by instruction 1. Thus, R_6 can be directly copied from R_1 without any intermediate operations. After these symbolic values are propagated, register coalescing and dead code elimination are further performed to eliminate instructions 2 and 4, which are not useful in the program any more.

6.2 Memory Disambiguation

After symbolic evaluation has been performed, the dependency graph depicted in Figure 6.3(c) is dramatically simplified. The only remaining dependencies are among memory loads and stores that have different base registers, which, however, may not actually alias at run time. Following the idea of [28], Tracy performs another major optimization called speculative memory disambiguation to eliminate these spurious memory dependencies. As illustrated in Figure 6.3(b), all memory references are divided into groups with different base registers. If the ranges of addresses covered by two groups are *disjoint*, memory references in one group are guaranteed not to alias with those in the other group. For each trace, Tracy selects certain number of large reference groups and inserts non-alias condition tests to each pair combination of them. If these tests are always passed during a profiling period, the trace is re-parallelized assuming the corresponding memory references do not alias. These tests also remain in the newly parallelized trace and abort the trace whenever they are not passed. As

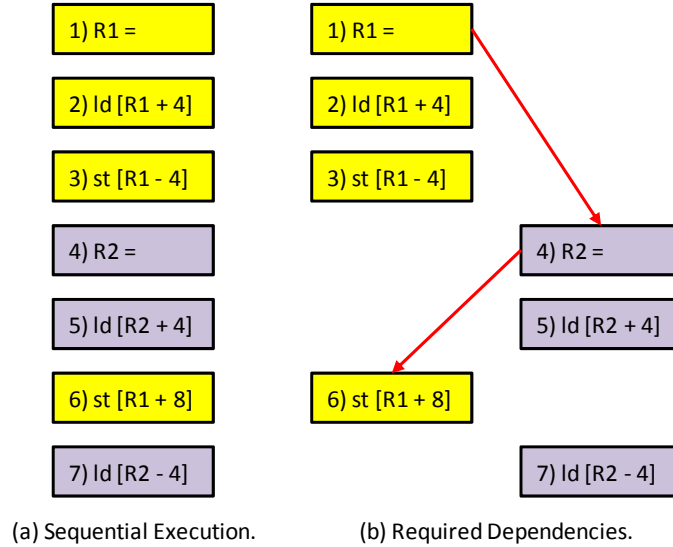


Figure 6.4: Tracy inserts extra dependencies to ensure that every monitored base register receives the correct value.

illustrated in Figure 6.3(c), when the address ranges of R_2 and R_3 are not overlapped, all spurious dependencies exist on the original CFG are eliminated.

One important prerequisite of correctly disambiguating memory references at the group granularity is that the tested base registers must always receive the correct values. If any base registers receive values that are actually produced by incorrect speculative memory accesses, these incorrect values may pass the non-alias condition tests that should have failed, leading to undetected errors during program execution. Thus, we require that memory references from two different groups can only start running out of order if both base registers have been produced and thus the corresponding test can be evaluated correctly. Figure 6.4(a) illustrates the sequential execution of seven instructions and Figure 6.4(b) shows two dependencies that are inserted by Tracy to ensure that every monitored base register receives the correct value. These extra dependencies ensure that the execution of instructions 2, 3, and 6 can only be interleaved with the execution of instructions 5 and 7 after base registers R_1 and R_2 have been produced. Thus, any violations of memory reference order can always be detected at the earliest possible time.

Accurate alias analysis is usually the key factor to enable effective program optimization and parallelization. DIS-based DBP techniques [3, 4, 5] usually rely on some centralized memory disambiguation unit to compare the address of each load or store instruction. Similarly, CFG-based DBP techniques [6, 7, 9] require special hardware support such as transactional memory to automatically detect violations of memory reference order. Tracy can insert all non-alias condition tests directly into the trace and treat them as normal instructions, which, however, greatly restricts the number

of memory reference groups that can be tested in order to maintain reasonable monitoring overhead. In fact, these tests have two characteristics that can make them run much more transparently and efficiently. First, each test either passes silently or aborts the candidate trace if the corresponding non-alias assumption has been proved incorrect. No value needs to be calculated and further used by other instructions in the trace. Thus, the entire process can be safely offloaded to some special functional units. Second, because all monitored base registers need to be tested against one another and the test is always an integer comparison, the single instruction multiple data (SIMD) computational model can be readily adopted to dramatically speed up the entire process.

Thus, within each slave cluster, an accelerator is also connected to the cores, which performs multiple integer comparisons per clock cycle. When each monitored base register is produced by the trace, its value and two offsets (as well as a bitmap that shows all other base registers against which it needs to be tested) are transferred to the accelerator, which calculates its address ranges and tests them against those of corresponding base registers whose information has already been received. This hardware support has two advantages: 1) the actual program execution is not affected, and 2) incorrect non-alias assumptions can be quickly detected to reduce mis-speculation penalty. Because only the information of monitored base registers need to be transferred between the processor and the accelerator, this technology has better scalability than that used by DIS-based DBP techniques, which requires every load and store instruction to be passed to the centralized memory disambiguation unit. On the other hand, disambiguating memory references at the group granularity introduces many false positives due to the approximation of memory addresses. This approximation, however, is inevitable in Tracy because once distributed to different threads, the sequential order of all memory references is lost. The TRIPS processor [19] redesigns the ISA by encoding the memory reference order into each load and store instruction, which greatly restricts the trace length and is not widely applicable to general many-core architectures.

6.3 Experimental Setup

Besides the experimental setup described in Section 5.5, we set the spill space to be 24 KB with one clock cycle access latency, because scratchpad memory has very simple caching logic. Prior research [123] has also shown that the chip area occupied by the scratchpad memory is less than the cache memory by 34%. The accelerator is configured to comprise four functional units, each of which can perform 32 integer comparisons per clock cycle. Thus, if the slave cluster is used to execute four candidate traces, each trace is assigned with one functional unit. We allow a maximum

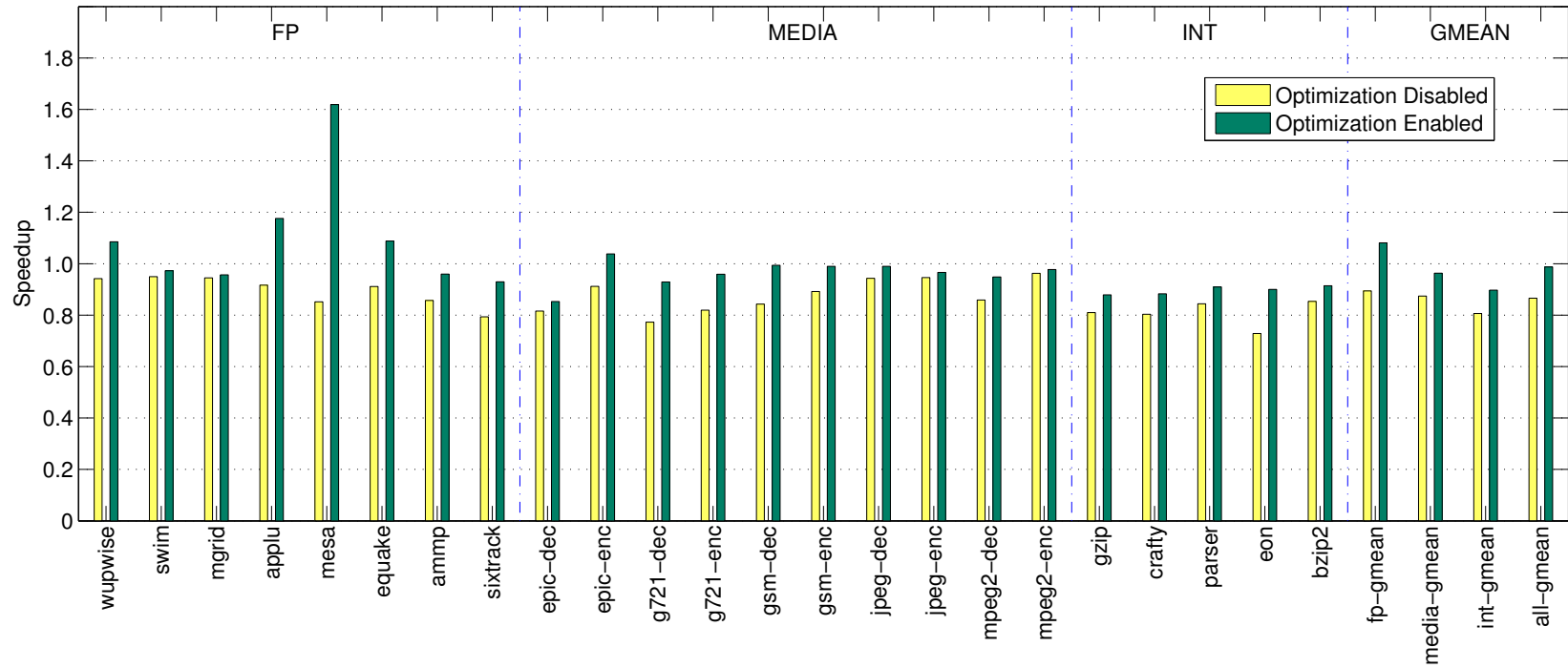


Figure 6.5: This figure shows the speedup of Tracy₃₄⁴-io2 when 1) optimization is disabled, and 2) optimization is enabled. Results are normalized to ST-io2.

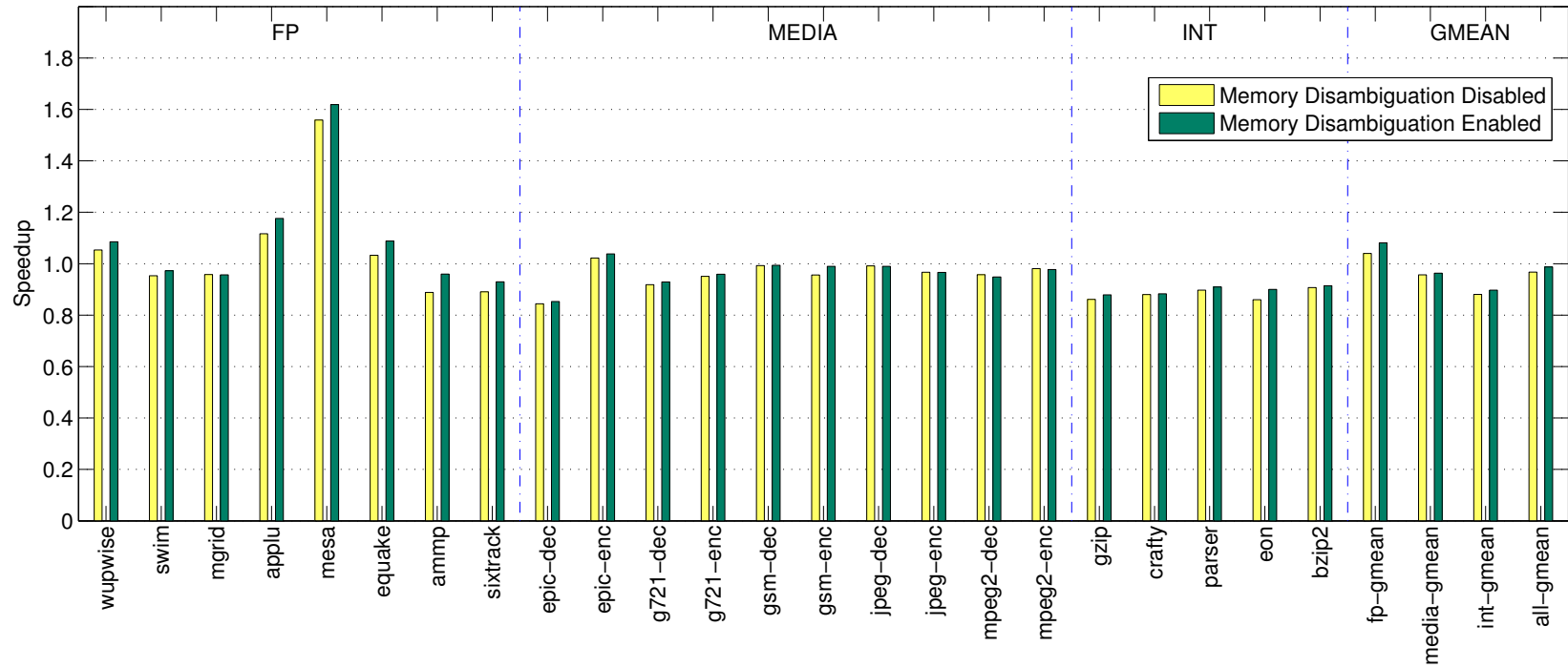


Figure 6.6: This figure shows the optimization-only speedup of Tracy₃₄⁴-io2 when 1) memory disambiguation is disabled, and 2) memory disambiguation is enabled. Results are normalized to ST-io2.

of 32 base registers to be monitored for traces that are shorter than 4K instructions and a maximum of 64 base registers to be monitored for longer traces. It only takes 46 clock cycles to test every pair combination of 32 base registers with one functional unit. This processing time is even negligible for executing the shortest trace, which, of course, has a much smaller number of base registers to be monitored. Similarly, it only takes 156 clock cycles to test every pair combination of 64 base registers with one functional unit, far fewer than those needed to execute traces longer than 4K instructions. Because one 64-bit single-clock-cycle integer comparator only needs 1,051 transistors to implement [130], the chip area needed by a total of 128 integer comparators provided by the entire accelerator is similar to that needed by about 88 32-byte cache lines.

We do not explicitly generate spill instructions and produce/consume instructions. Every register that needs to access the spill space or the synchronization array is marked on the original instruction, and the access latency is directly inserted into the SESC simulation framework. Since traces only contain straightline instruction sequences, we assume a simple hardware prefetcher which continuously fetches the entire trace into the L1 instruction cache. If memory disambiguation is enabled in the experiment, we further assume that the corresponding profiling information has been obtained in prior execution of the same traces.

6.4 Experimental Results

Figure 6.5 illustrates the performance of Tracy₃₄⁴-io2 when 1) optimization is disabled, and 2) optimization is enabled. Results are normalized to ST-io2. Thus, the first case actually shows the pure overhead of our framework by executing non-optimized traces sequentially. The slowdown is 0.86x, averaged over all benchmarks. Although optimization increases the program performance with pure overhead by a factor of 1.14x, the overall speedup is only achieved from four floating point benchmarks (*wupwise*, *applu*, *mesa*, *equake*) and one media benchmark (*epic-enc*). Trace-based DBO has achieved better speedups in prior research [18, 20, 21, 32], however, the overhead of program state transfer to and from the candidate traces impose a significant challenge on Tracy. One interesting observation is that both *swim* and *mgird* have very long traces, but the performance increase due to optimization is almost negligible. In fact, a large number of redundant or dead instructions are eliminated from the constructed traces. For both benchmarks, the reason of the poor optimization performance is that many L2 cache misses are encountered during program execution, which becomes the major bottleneck. On the contrary, optimization opportunities are transferred to great speedups in *applu* (1.17x) and *mesa* (1.61x). Figure 6.6 compares the optimization-only perfor-

mance of Tracy₃₄⁴-io2 when 1) memory disambiguation is disabled, and 2) memory disambiguation is enabled. The same trend that memory disambiguation only slightly increases the optimization performance can be observed in almost all benchmarks.

The above experimental results may incorrectly lead to the conclusion that both symbolic evaluation and memory disambiguation are not critical to the general performance of Tracy. This is true when only optimization is performed because program execution is still restricted to one single core. However, the functionality of these optimizations is not only to directly produce speedups, but more importantly, to prepare the code for future parallelization. Both optimizations actually focus on eliminating unnecessary data dependencies (both register and memory) or increasing data dependency lengths, exposing more parallelism opportunities. Although these effects are not manifest when only optimization is performed, they are extremely critical to the overall parallel performance, which will be evaluated in Section 7.5.5.

6.5 Summary

In this chapter, we describe two optimizations, symbolic evaluation and memory disambiguation, which are specially customized for atomic traces. Because symbolic evaluation performs path-sensitive program analysis and data-flow information is based on symbolic values instead of lexical names, it performs more precise program analysis than traditional data-flow analysis. Disambiguating memory references at the group granularity introduces many false positives due to the approximation of memory addresses. This approximation, however, is inevitable in Tracy because once distributed to different threads, the sequential order of all memory references is lost.

Although optimization alone is generally not enough to speed up applications (0.98x slowdown on average) due to the overhead of program state transfer, it sets a solid foundation for future parallelization by dramatically releasing dependent constraints.

Chapter 7

Trace Parallelization

Although DIS- and CFG-based DBP techniques are complementary to each other, no prior research has tried to implement both technologies under a unified system. Such a system can achieve large speedups from code regions that contain coarse-grained LLP or TLP, and exploit ILP from the remainder of the program. This combination is quite vital. As Amdahl's Law shows, even a small fraction of non-parallelizable code can drastically inhibit overall speedups.

This chapter describes the parallelization strategy adopted by Tracy, which leverages traces as the unified representation of program execution to exploit both LLP and ILP. Tracy does not intend to develop totally innovative parallelization algorithms, but to customize off-the-shelf algorithms to make them suitable for parallelizing atomic traces.

7.1 Exploiting ILP

Tracy adopts the traditional list scheduling algorithm [36] to exploit ILP by partitioning and scheduling instructions among different cores. First, the dependency graph is built. Next, priorities are assigned to each instruction in the graph based on its ALST. Instructions with smaller ALST are scheduled first. Finally, the list scheduler places instructions into the schedule cycle by cycle, starting from cycle zero. Any instruction whose operands have been computed at cycle X is a candidate to be scheduled at that cycle, within the confines of hardware availability. The priorities computed in the prior step are used to determine which ready instruction to schedule, with ties being broken arbitrarily. For list scheduling, the more accurate the machine model is, the better performance can be achieved because the predicted schedule more precisely matches the exact execution order of instructions on the real machine. Tracy only considers two major factors in the machine model:

1) the issue width, and 2) the number of functional units and the corresponding execution latency. The list scheduling algorithm becomes much more complicated when considering more factors in the machine model, which may not be appropriate to be performed at run time.

The traditional list scheduling algorithm only reorders instructions on the same core to fully utilize the hardware. Tracy, however, also has to partition instructions among different cores. Thus, we make two modifications to the original algorithm. First, inter-core communication overhead is inserted if the value needs to be transferred from one core to another. Second, all memory loads and stores with the same base register are assigned to the same core so as to minimize cache coherence traffic, which may greatly hurt the program performance [131]. Furthermore, in order to fully utilize the L1 data cache of all cores, memory loads and stores with different base registers are evenly assigned to the parallelized threads.

7.2 Exploiting LLP

Unlike ILP that can be exploited in all traces, LLP can only be exploited in traces that comprise multiple loop iterations. As in trace construction, Tracy starts to parallelize the outermost loop by evenly distributing its iterations to the available number of cores. If the outer loop is not parallelizable, Tracy then starts to parallelize the inner loops. Even if the outer loop is parallelizable, those iterations that are left from the even distribution are further parallelized using the inner loops if possible. Tracy performs two major code transformations, accumulator expansion and dependent code motion to handle loop-carried dependencies that are not eliminated by prior optimization, which are essential to make more loops parallelizable.

An accumulator is a variable that is repeatedly updated during every iteration of the loop. Figure 7.1(a) illustrates the source code of a small loop with eight iterations, in which *sum* is an accumulator. Unfortunately, when the entire loop is formed into a trace and partitioned into two threads by exploiting LLP, the code is almost serialized because each update of *sum* cannot proceed until the prior update is performed. Figure 7.1(b) depicts the two threads with the offending dependency, in which *sum* is stored in R_1 . Figure 7.1(c) shows the same parallelized trace after accumulator expansion. The accumulator *sum* is replaced by two different variables, stored in R_1 and R_2 , respectively. These two variables become private accumulators in different threads so that the offending dependency is broken. Each private accumulator must be initialized to zero. Furthermore, an extra instruction must be inserted to calculate the final value of *sum* by adding two private accumulators together. Although one dependency still exists, the value from the first

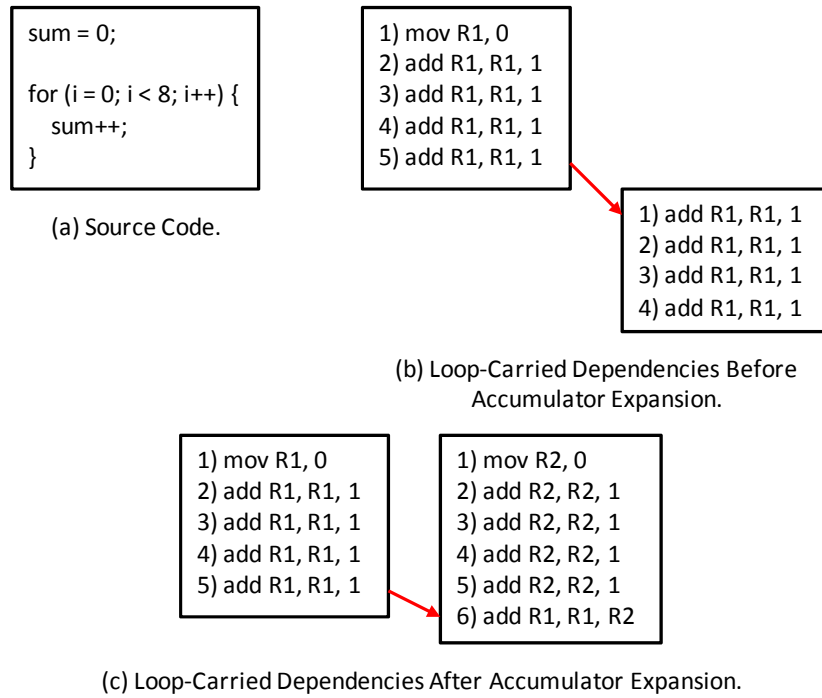


Figure 7.1: Accumulator expansion replaces the single shared accumulator with multiple private accumulators.

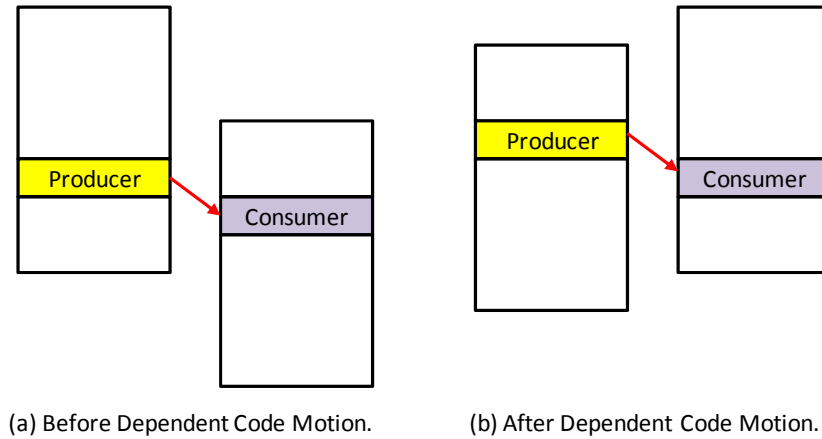


Figure 7.2: Dependent code motion pushes every producer to be executed earlier and every consumer to be executed later.

thread is only needed at the end of the second thread, making the trace quite parallelizable. It is extremely difficult to find all cases where accumulator expansion can be applied. Tracy searches for opportunities by checking every *add* or *addu* instruction in MIPS.

Dependent code motion is a much more general technology than accumulator expansion because it can be performed on any dependencies between two threads. Figure 7.2(a) illustrates the original synchronization between the producer and consumer when the trace is partitioned into two threads.

Tracy reschedules the producer (with all its predecessors) to be executed as early as possible and the consumer (with all its successors) to be executed as late as possible. As depicted in Figure 7.2(b), the overlapped part of the two threads becomes much larger, leading to better parallel performance. Dependent code motion is similar to code prematerialization [6, 132], which, however, re-calculates the live-in values in the logically later thread instead of pushing them to be calculated as soon as possible in the logically earlier thread.

7.3 Combining ILP and LLP

Tracy selects the optimal parallelization strategy at the trace level. It first parallelizes the trace by exploiting LLP, which has the potential to produce larger speedups. If limited LLP exists, however, Tracy extracts ILP from the trace instead. During preliminary experiments, we have observed that exploiting LLP typically only leads to better parallel performance than exploiting ILP if two requirements are satisfied simultaneously. First, the outermost loop in the trace should be itself parallelizable. Only parallelizing the inner loops would make a large portion of the trace run sequentially. Second, the outermost loop in the trace should be nearly a DOALL loop [56] with zero or very few loop-carried dependencies. When a relatively large number of loop-carried dependencies exist, exploiting ILP usually generates larger speedups because instructions can be scheduled at finer granularity. Based on the above observations, Tracy uses a simple heuristic to determine the optimal parallelization strategy. After all LLP has been extracted from the trace, Tracy divides the sequential execution time by the parallel execution time, assuming that each instruction takes one clock cycle to run. If the estimated speedup is greater than or equal to 1.75 for 2-way parallelization, LLP is more appropriate to be exploited in the trace. Otherwise, Tracy abandons the parallelized trace and starts to exploit ILP instead. Similarly, the strategy selection threshold for 4-way and 8-way parallelization is 3.5 and 7.25, respectively.

Because Tracy performs the list scheduling algorithm before register allocation, it is traditionally known to generate much more register spills [33]. Similarly, dependent code motion would also extend the live range of certain registers. However, this problem is substantially alleviated in Tracy due to three reasons. First, because the trace is partitioned among multiple cores, the number of registers is also increased proportionally. Second, if several instructions on the same core need the value in the same entry of the synchronization array, each instruction has to fetch the value separately. This prevents the value from being stored locally, greatly reducing the register pressure. Third, the dedicated spill space made of scratchpad memory causes reads and writes of spilled registers to

INT/FP Issue Queue Entry	16/16
ROB Entry	48
Load/Store Queue Entry	12/12

Table 7.1: Extra architectural parameters of 2-issue OoO cores.

always have very low latency, dramatically reducing the overhead caused by register spills. Although the phase ordering problem of instruction scheduling and register allocation is not the topic of this dissertation, prior research [133, 134] has proposed several algorithms to reduce register pressure without affecting the ultimate schedule length.

7.4 Experimental Setup

Besides the experimental setup described in Sections 5.5 and 6.3, this section also evaluates the performance of Tracy on different types of OoO cores. A 2-issue OoO core shares all the architectural parameters of a 2-issue IO core listed in the third column of the first row in Table 5.3, with extra parameters related to OoO execution listed in Table 7.1, separately. A 4-issue OoO core has two times the amount of certain resources as a 2-issue OoO core, including the fetch/issue/retire width, extra INT/FP registers for renaming, INT/FP issue queue entry, ROB entry, and load/store queue entry. In all configurations, the L1 caches, L2 cache, interconnect, memory subsystem, and Tracy remain exactly the same.

We use the Wattch [135] and CACTI [136] models attached to the SESC simulation framework to calculate dynamic power consumption and assume that aggressive clock gating is supported in all processor structures. Access energy of the spill space and the synchronization array is first modeled as access energy of a similar cache structure and then roughly reduced using the data published by [123]. We only consider energy for trace construction and prediction at possible program points (i.e., heads of active code structures) and assume that these points can be dynamically marked on the original binary executable. When Tracy is operating, idle cores consume 5% of their original dynamic power [137]. However, this dynamic power leakage of idle cores is not counted in single-threaded execution to act as a challenging comparison base.

7.5 Experimental Results

Although parallelization is the final and major step to speed up single-threaded software, the performance of Tracy is holistically dependent on trace construction, trace prediction, and trace opti-

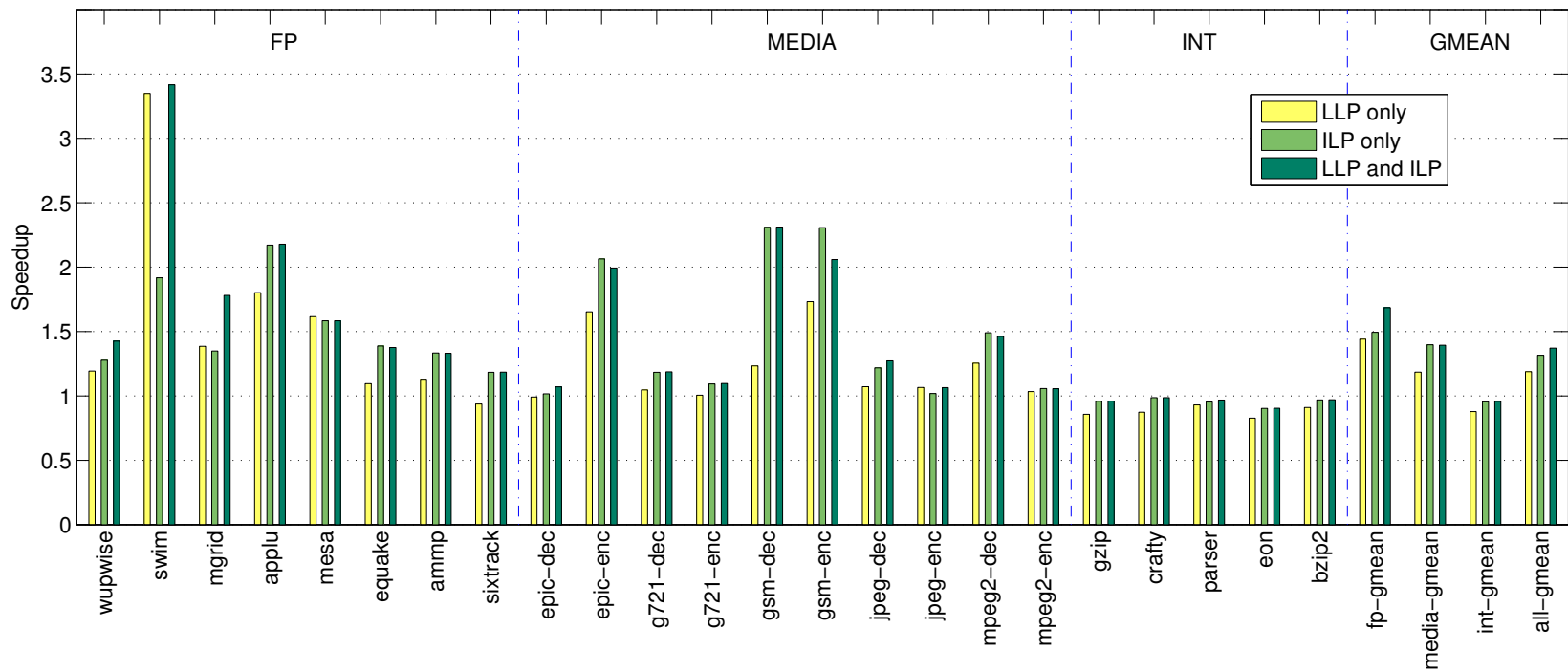


Figure 7.3: This figure shows the speedup of Tracy₃₄-io2 when it is configured to exploit 1) LLP only, 2) ILP only, and 3) both LLP and ILP. Results are normalized to ST-io2.

Benchmark		Exec. on LLP %
FP	wupwise	2.51 %
	swim	84.87 %
	mgrid	33.64 %
	applu	0.09 %
	mesa	0.00 %
	equake	0.03 %
	ammp	0.00 %
	sixtrack	0.47 %
MEDIA	epic-decode	10.12 %
	epic-encode	1.18 %
	g721-decode	0.00 %
	g721-encode	0.00 %
	gsm-decode	0.28 %
	gsm-encode	49.05 %
	jpeg-decode	19.91 %
	jpeg-encode	50.60 %
	mpeg2-decode	29.90 %
	mpeg2-encode	32.59 %
INT	gzip	1.59 %
	crafty	0.03 %
	parser	19.29 %
	eon	0.33 %
	bzip2	0.30 %

Table 7.2: This tables shows the percentage of instructions executed by the unmodified program that are covered by correctly predicted traces in which LLP is exploited over instructions that are covered by any correctly predicted traces.

mization, whose effectiveness has been separately evaluated in Sections 5.5 and 6.4. This section, on the other hand, mainly evaluates Tracy as an entire system. We first compare the performance of Tracy₃₄⁴-io2, our major experimental system, when it is configured to adopt different parallelization strategies. The IO cores are also upgraded to more advanced OoO cores to test its generality. We then compare the performance of Tracy with the performance of existing DIS- and CFG-based DBP techniques. For a comprehensive evaluation, Tracy is thoroughly evaluated afterward using different system configurations and architectural parameters. Finally, we analyze the performance of Tracy in detail by isolating the overheads and benefits caused by different factors.

7.5.1 Overall Performance using Different Parallelization Strategies

Figure 7.3 illustrates the speedup of Tracy₃₄⁴-io2 when it is configured to exploit 1) LLP only, 2) ILP only, and 3) both LLP and ILP. Results are normalized to ST-io2. When hybrid parallelism is exploited in the third case, Table 7.2 shows the percentage of instructions executed by the unmodified program that are covered by correctly predicted traces in which LLP is exploited over instructions

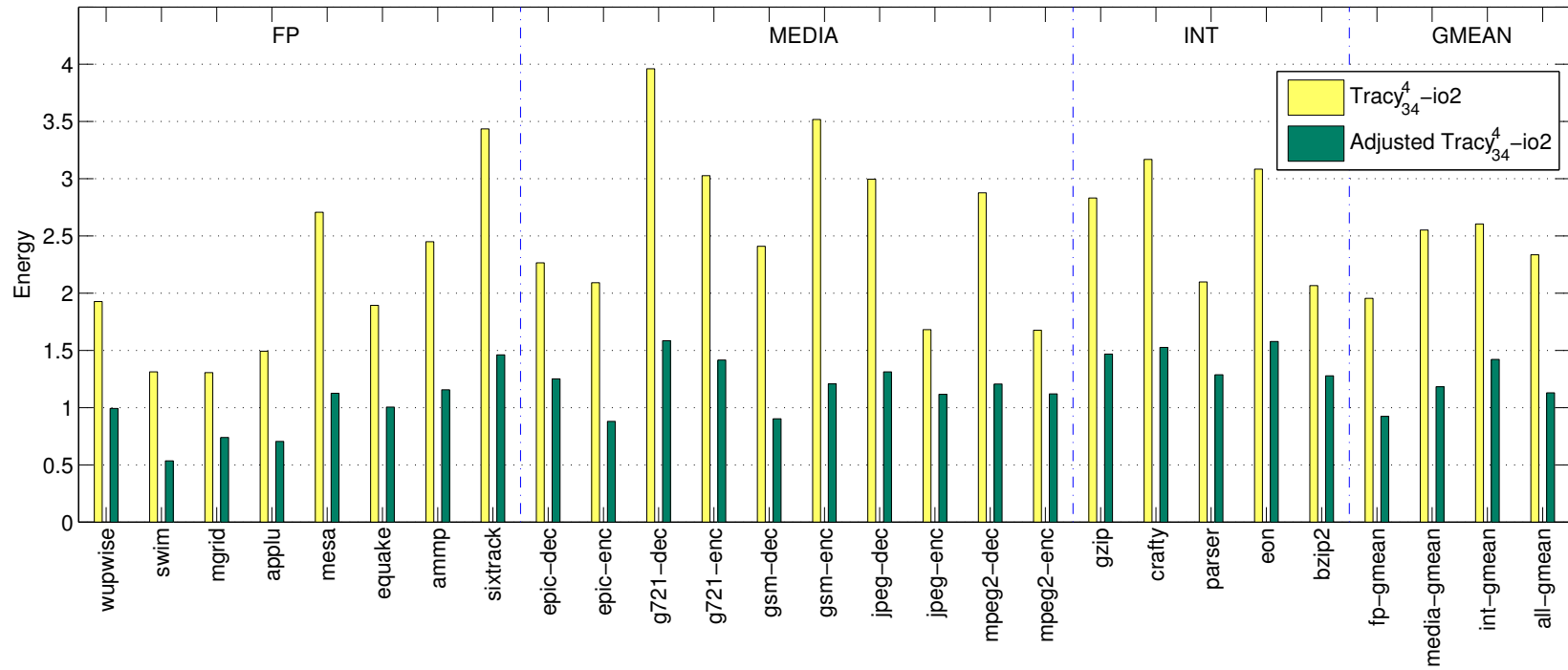


Figure 7.4: This figure shows the energy consumption of Tracy₃₄⁴-io2 when it is configured to exploit both LLP and ILP. Results are normalized to ST-io2. The adjusted energy consumption is achieved by counting in “system leakage” from other machine components and power supply inefficiencies.

that are covered by any correctly predicted traces. Generally speaking, only exploiting ILP produces greater speedups than only exploiting LLP, which is not surprising because it can schedule instructions at finer granularity. While the ILP-only speedup is 1.31x, averaged over all benchmarks, the LLP-only speedup is only 1.18x. The three obvious outliers are *swim*, *mgrid*, and *jpeg-enc*, all of which comprise highly parallelizable loops with zero or very few loop-carried dependencies. The specific situation, however, is quite different for these three benchmarks. In *swim*, LLP can be extracted from 81.94% of dynamic instructions, leading to the great speedup of 3.34x. In *mgrid*, however, only 39.35% of dynamic instructions contain exploitable LLP so that the speedup is only 1.38x, slightly better than the speedup achieved by only exploiting ILP. The power of LLP is dramatically diluted by those sequentially executed instructions. The same problem is exaggerated in *jpeg-enc*, in which only 24.75% of dynamic instructions can be parallelized by only exploiting LLP. Thus, the performance increase over ILP-only speedup is negligible. The final outlier is *mesa*, which contains no LLP that can be effectively extracted. In this case, exploiting ILP actually decreases the program performance that has been dramatically increased by optimization.

For five out of eight floating point benchmarks with very limited LLP, the speedup achieved by exploiting both LLP and ILP is almost the same as the speedup achieved by only exploiting ILP. On the other extreme, although LLP is extracted from 84.87% of the instructions covered by correctly predicted traces in *swim*, ILP in the remaining instructions still improves the program performance by another factor of 1.02x. A more balanced distribution exists in *mgrid*, in which LLP is extracted from 33.64% of the instructions covered by correctly predicted traces. Thus, the hybrid speedup (1.77x) is much better than the speedup achieved by only exploiting LLP (1.38x) or ILP (1.34x). More interestingly, LLP is only exploited in 2.51% of the instructions covered by correctly predicted traces in *wupwise*, but the hybrid speedup is 1.11x of the ILP-only speedup. The potential reason is that the corresponding code region has higher CPI so that the relative importance of these instructions is dramatically increased. For media benchmarks, Tracy actually predicts that more traces should be parallelized by extracting LLP. The relatively shorter trace length, however, reduces the problem of local optimization in the list scheduling algorithm, making the ILP-only speedup typically better. Over-estimating the power of LLP, Tracy mispredicts the optimal parallelization strategy in several cases, including *epic-enc*, *gsm-enc*, and *mpeg2-dec*. Because the hybrid parallelization strategy produces the best speedup averaged over all benchmarks, we only evaluate this strategy in the remaining of this section.

The hybrid speedup of floating point benchmarks can exceed 1.4x for five out of eight applications, with an average number of 1.68x. Similarly, an average speedup of 1.39x can also be achieved

	FP		MEDIA	
	Improved #	Speedup	Improved #	Speedup
Tracy ₃₄ ⁴ -io2	8 / 8	1.68x	10 / 10	1.39x
Tracy ₃₄ ⁴ -ooo2	6 / 8	2.03x	4 / 10	1.69x
Tracy ₃₄ ⁴ -ooo4	6 / 8	1.70x	4 / 10	1.30x

Table 7.3: This table summarizes the performance of 1) Tracy₃₄⁴-io2, 2) Tracy₃₄⁴-ooo2, and 3) Tracy₃₄⁴-ooo4. Results are normalized to ST-io2, ST-ooo2, and ST-ooo4, respectively. Two metrics are evaluated for each category of benchmarks: 1) the number of programs with improved performance, and 2) the speedup averaged over programs with improved performance.

for media benchmarks. However, Tracy actually slows down all integer benchmarks, although to a very small extent. This is not surprising because integer programs normally have more complicated control flows that are hard to predict and pointer-based memory references that are hard to disambiguate. Furthermore, the average trace length for integer benchmarks is only 95, hardly exposing any distant parallelism opportunities. Some media benchmarks (e.g., *epic-dec*, *g721-dec*, *g721-enc*) or even floating point benchmarks (e.g., *sixtrack*) also have relatively short traces, which, however, contain more ILP so that mediocre speedups can still be achieved. Because Tracy generally does not produce reasonable speedups for integer benchmarks, we only evaluate floating point and media benchmarks in the remaining of this section.

Although speeding up most benchmarks, Tracy does consume more energy due to multi-trace execution and inter-core synchronization. Figure 7.4 illustrates that Tracy consumes 1.95x and 2.55x energy for floating point and media benchmarks, respectively. As described in Section 4.2, however, the CPU chip is only a small fraction of overall energy consumption. Large DRAM, disk, power supply inefficiencies, etc. are also major factors that impose a constant background “system leakage” while the machine is awake. If we assume the CPU chip consumes 34% of the system power [138] and the machine wastes 30% of the power it consumes due to supply inefficiencies [139], the speedup achieved by Tracy means that the average energy consumed is actually reduced to 0.92x (floating point benchmarks) and 1.18x (media benchmarks) of that consumed by single-threaded execution. When the speedup is large enough, Tracy actually saves energy (sometimes dramatically) in several cases, including *swim*, *mgrid*, *applu*, *epic-enc*, and *gsm-dec*.

7.5.2 Upgrading to OoO Cores

Although we believe that IO cores are more suitable to be the building block of throughput-oriented many-core architectures in the future, we also evaluate Tracy by upgrading 2-issue IO cores to 2-issue OoO cores (Tracy₃₄⁴-ooo2) and 4-issue OoO cores (Tracy₃₄⁴-ooo4). The other parts of the

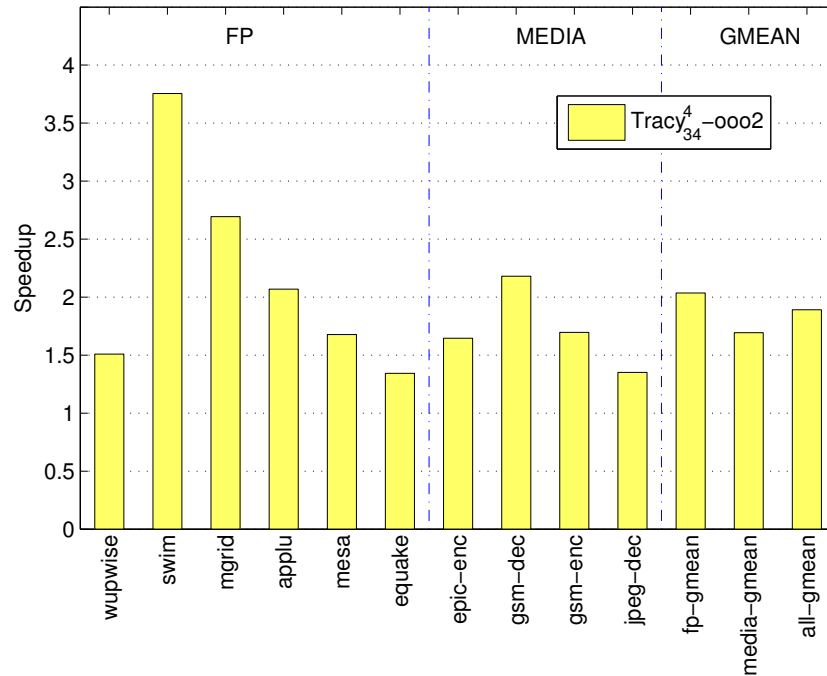


Figure 7.5: This figure shows the speedup of Tracy₃₄⁴-ooo2 when it is configured to exploit both LLP and ILP. Results are normalized to ST-ooo2.

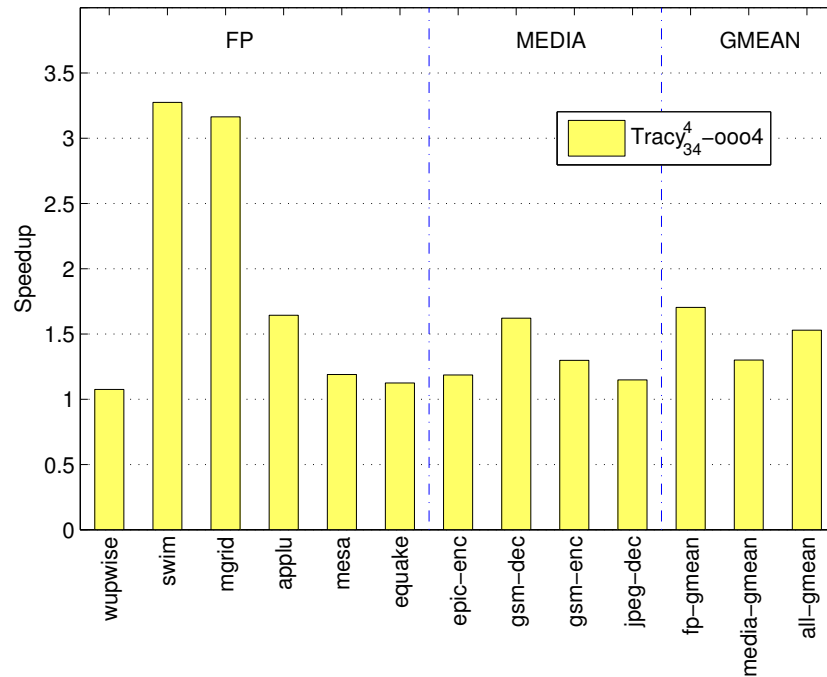


Figure 7.6: This figure shows the speedup of Tracy₃₄⁴-ooo4 when it is configured to exploit both LLP and ILP. Results are normalized to ST-ooo4.

system remain unchanged. The results are depicted in Figures 7.5 and 7.6, respectively. Table 7.3 summarizes the performance of Tracy using all three configurations and speedup is normalized to the

corresponding single-threaded execution.

The performance of Tracy is quite sensitive to core type. When using OoO cores instead of IO cores, Tracy can speed up six out of eight floating point benchmarks and four out of ten media benchmarks. These benchmarks that have speedups all share one common characteristic that the average trace length is larger than 500 instructions. Based on this observation, Tracy can be configured to always construct traces, but only parallelize and predict them when the average trace length exceeds the threshold. As a result, Tracy becomes a “hippocratic” technique that never hurts the performance achieved by single-threaded execution.

For most of the benchmarks that can benefit from Tracy using OoO cores, the speedup still decreases dramatically as the issue width increases. The reason is that when the trace is partitioned into different threads by extracting ILP, the amount of ILP that remains on each core is decreased simultaneously. As a result, the OoO core extracts much less ILP from each thread than it does from single-threaded execution, leading to diminishing returns. When LLP is the major type of parallelism, however, the benchmark is much less sensitive to core type. For example, while the speedup of Tracy₃₄⁴-io2 is 3.41x for *swim*, the speedup of Tracy₃₄⁴-ooo2 is even increased to 3.75x and the speedup of Tracy₃₄⁴-ooo4 is only decreased to 3.27x.

7.5.3 Comparing to DIS- and CFG-Based DBP Techniques

Figure 7.7 illustrates the performance comparison of Tracy with Core Fusion [3], one representative of existing DIS-based DBP techniques. The experimental data of Core Fusion is directly obtained from the cited publication. Both systems are implemented in the SESC simulation framework and share the same architectural parameters so that the comparison is relatively fair. Because Core Fusion combines four 2-issue OoO cores to provide an increased issue width, our system is configured as both Tracy₃₄⁴-ooo2 and Tracy₆₆⁸-ooo2 accordingly. Core Fusion relies on centralized hardware to support collaborative fetch, renaming, memory disambiguation, and commit, so that we believe it is hard to be scaled to combine eight cores simultaneously.

Although not included in Figure 7.7, Core Fusion can actually achieve an average speedup of 1.3x for integer benchmarks, greatly outperforming the performance of Tracy. One major reason is that a large portion of dynamic instructions is not covered by parallelized traces in integer benchmarks, which, however, can benefit from Core Fusion through the entire program execution. Furthermore, integer benchmarks typically have complicated control flows, resulting in short traces (95 instructions

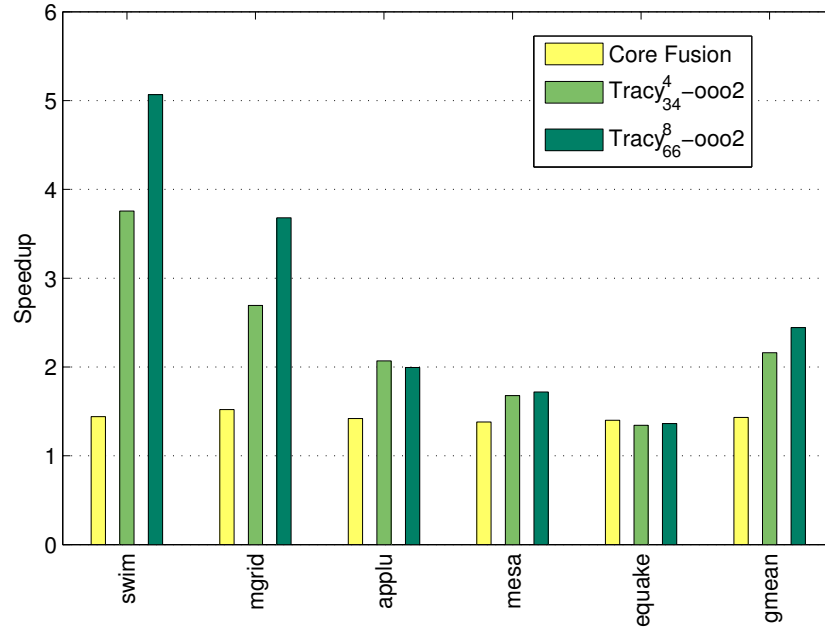


Figure 7.7: This figure compares the performance of Core Fusion, one representative of DIS-based DBP techniques, with Tracy₃₄⁴-ooo2 and Tracy₆₆⁸-ooo2 when they are configured to exploit both LLP and ILP. Results are normalized to ST-ooo2.

on average). These traces do not contain enough parallelism to overcome the overheads of inter-core synchronization and program state transfer.

For floating point benchmarks, however, Tracy can achieve an average speedup of 2.16x (4-way parallelization), 1.51x better than the speedup achieved by Core Fusion. When Tracy performs 8-way parallelization, the speedup is further increased to 2.44x. The relative performance of Tracy and Core Fusion is highly dependent on the availability of distant parallelism. If only local parallelism exists, the instruction issue mechanism of OoO execution is much more efficient than pre-inserted synchronizations. Core Fusion outperforms Tracy in *equake*, which mainly contains ILP. Although *equake* has an average trace length of 600 instructions, the lack of distant parallelism greatly restricts the capability of Tracy to exploit opportunities that do not exist in the hardware instruction window. An opposite example is *applu*, whose long traces (4,251 instructions on average) expose a large amount of distant ILP opportunities so that the performance of Tracy is 1.45x (4-way parallelization) and 1.40x (8-way parallelization) better than the performance achieved by Core Fusion. On the other hand, Tracy can extract a large amount of LLP from both *swim* and *mgrid*, resulting in much greater speedups than the speedup achieved by Core Fusion. For example, the speedup of *swim* achieved by Tracy is 3.75x based on 4-way parallelization, while it is only 1.44x achieved by Core Fusion. When Tracy performs 8-way parallelization, the speedup is further increased to 5.06x, demonstrating quite

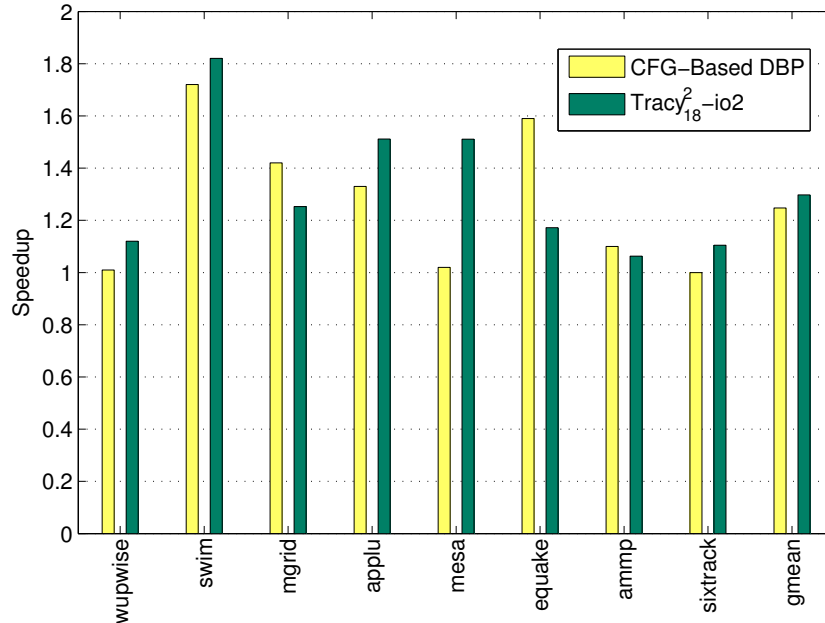


Figure 7.8: This figure compares the performance of CFG-based DBP with Tracy²₁₈-io2 when it is configured to exploit both LLP and ILP. Results are normalized to ST-io2.

good scalability that is hardly reachable by any DIS-based techniques.

Figure 7.8 illustrates the performance comparison of Tracy with one representative CFG-based DBP technique [6]. The experimental data (transactional memory is implemented at cache line granularity) of CFG-based DBP is directly obtained from the cited publication. Because CFG-based DBP performs 2-way parallelization using IO cores, our system is configured as Tracy²₁₈-io2. Although the absolute performance is less meaningful because different simulation frameworks are used, we believe the overall trend is widely applicable.

Although the average speedup achieved by Tracy is only 1.04x better than the speedup achieved by CFG-based DBP, Tracy can speed up all floating point benchmarks while CFG-based DBP fails to parallelize three out of eight applications at all. The reason is that when no LLP is exploitable, CFG-based DBP lacks the capability to schedule instructions at finer granularity. We believe that general applicability is very important for the success of any DBP techniques. On the other hand, CFG-based DBP has the capability to exploit more LLP because it can parallelize any loops instead of only those restricted by trace length. When enough LLP is available, CFG-based DBP typically produces great speedups. The most obvious examples are *mgrid* and *equake*, in which CFG-based DBP outperms Tracy by a factor of 1.13x and 1.35x, respectively.

7.5.4 Changing System Configurations and Architectural Parameters

In order to test the performance stability of Tracy, we conduct four sets of sensitive analysis by varying 1) the number of candidate traces that can be executed simultaneously, 2) the number of parallel threads that each trace is decomposed into, 3) the access latency of the synchronization array, 4) the transfer delay of the backbone bus, and 5) the access latency of the spill space.

Figure 7.9 shows the performance of Tracy by varying the number of candidate traces that can be executed simultaneously. The average speedup for media benchmarks is more sensitive than the speedup for floating point benchmarks to the maximum number of candidate traces, because they typically speculate larger number of candidate traces on average. However, the capability to speculate more candidate traces simultaneously does not necessary lead to better parallel performance. In many cases, parallelization actually slows down the program when the corresponding code region is covered by short traces. When the number of available cores is reduced, the same code region is considered non-parallelizable by Tracy and starts to run sequentially, leading to improved overall performance. The most interesting example is *gsm-enc*. When only two candidate traces are supported, some code regions are not parallelized by Tracy because the speculation accuracy is relatively low. When four candidate traces are supported, these code regions are just above the threshold to be parallelized, but the traces are so short that the the program is actually slowed down. When eight candidate traces are supported, the traces that cover the same code regions become longer, which finally has positive contributions to the overall performance.

However, a more important takeaway from Figure 7.9 is that Tracy does not require a large number of cores to operate effectively. Using only ten cores, an average speedup of 1.66x and 1.32x can still be achieved for floating point and media benchmarks, respectively. When using 34 cores, the speedup for floating point benchmarks is only increased by another factor of 1.01x and the speedup for media benchmarks is only increased by another factor of 1.05x. As described in Section 5.5, it is more economical for Tracy to only use part of the available cores, which are just enough to achieve near-optimal performance with much less energy consumption.

Figure 7.10 shows the performance of Tracy by varying the number of parallel threads that each trace is decomposed into. Increasing the parallelization width generally benefit all benchmarks because more hardware resources are available to execute the same candidate trace. For 2-way parallelization, the speedup is only 1.21x, averaged over all benchmarks. It is greatly increased to 1.51x and 1.67x when 4-way and 8-way parallelization is performed. The actual speedup for each benchmark, however, is highly dependent on the amount of existing distant parallelism. For

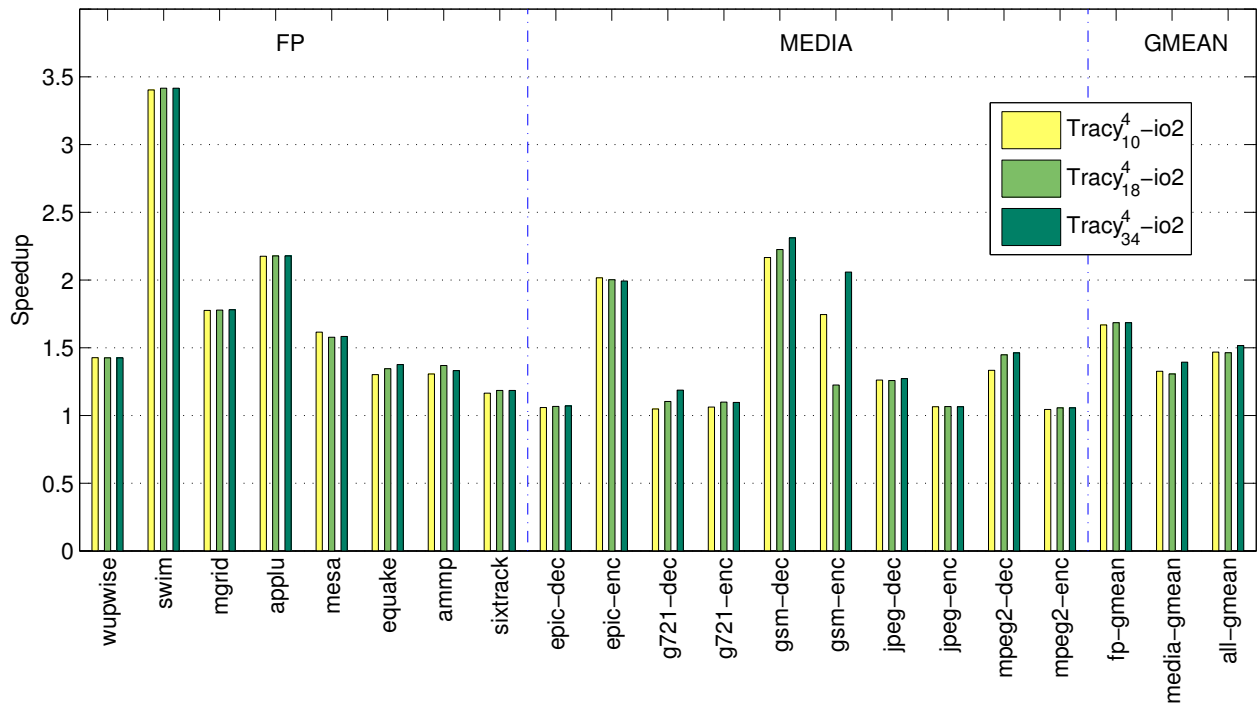


Figure 7.9: This figure shows the speedup of 1) Tracy₁₀⁴-io2, 2) Tracy₁₈⁴-io2, and 3) Tracy₃₄⁴-io2 when they are configured to exploit both LLP and ILP. Results are normalized to ST-io2.

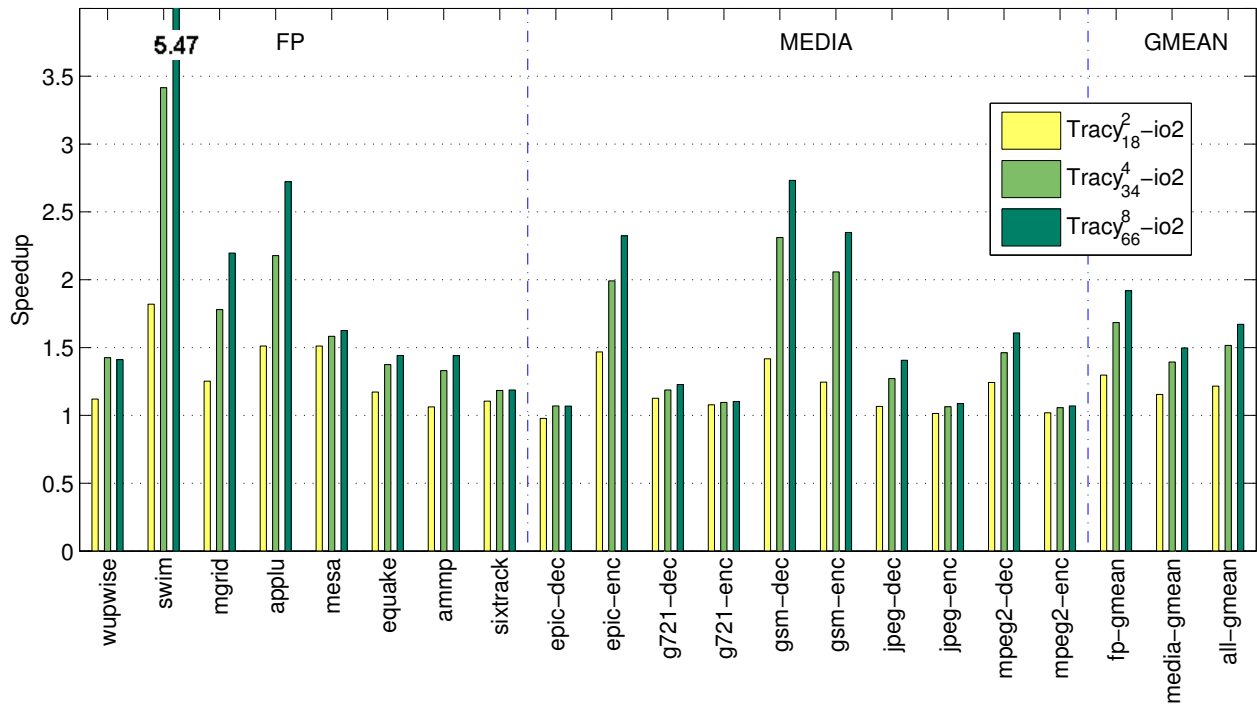


Figure 7.10: This figure shows the speedup of 1) Tracy₁₈²-io2, 2) Tracy₃₄⁴-io2, and 3) Tracy₆₆⁸-io2 when they are configured to exploit both LLP and ILP. Results are normalized to ST-io2.

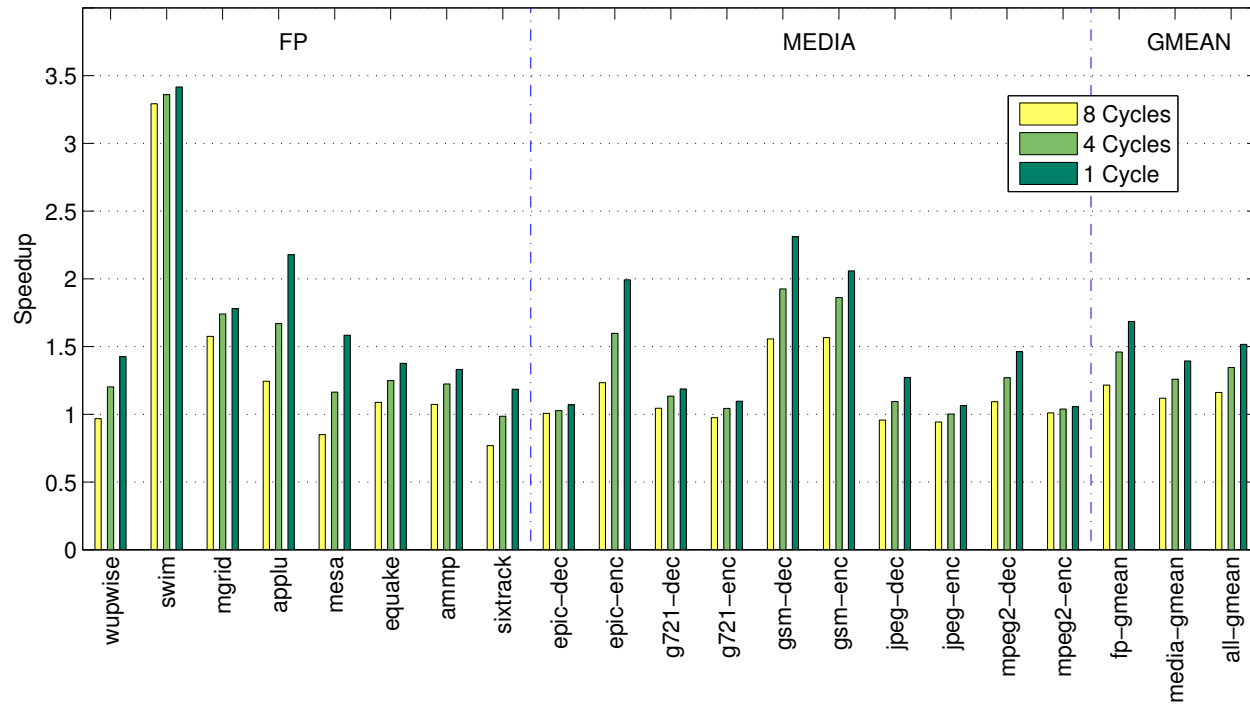


Figure 7.11: This figure shows the speedup of Tracy₃₄-io2 when the synchronization array has access latency of 1) 8 clock cycles, 2) 4 clock cycles, and 3) 1 clock cycle. The system is configured to exploit both LLP and ILP. Results are normalized to ST-io2.

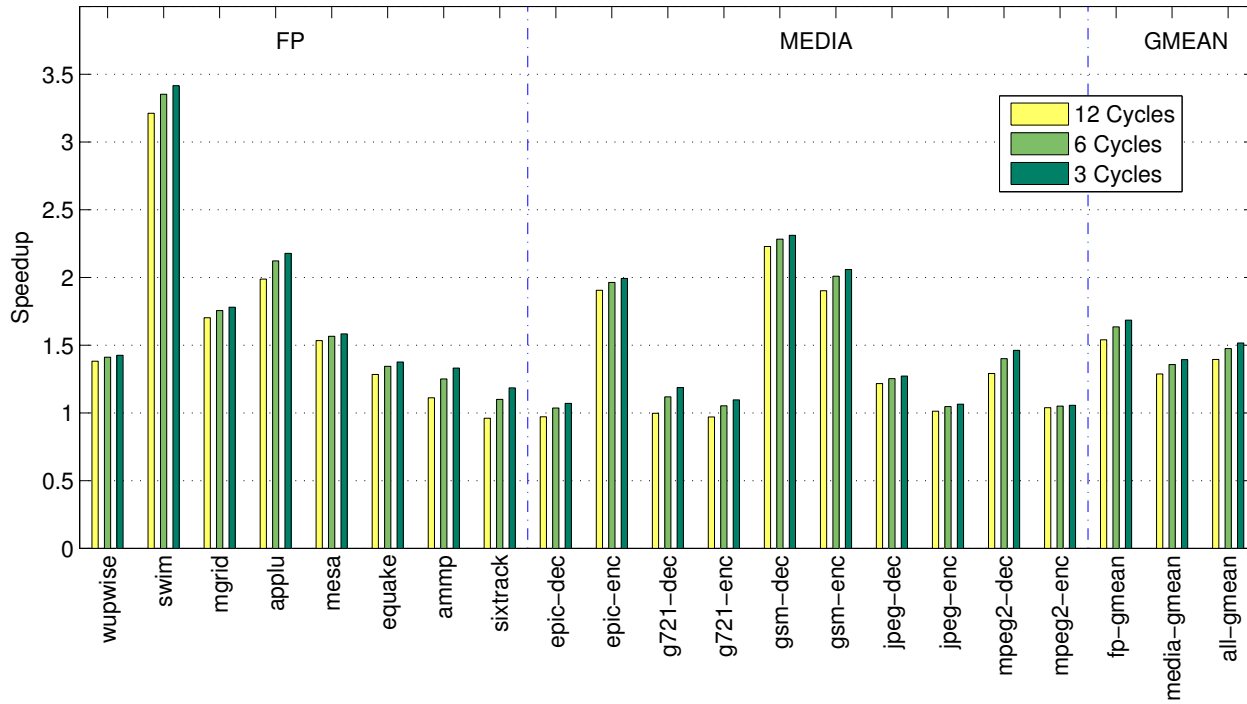


Figure 7.12: This figure shows the speedup of Tracy₃₄-io2 when the backbone bus has transfer delay of 1) 12 clock cycles, 2) 6 clock cycles, and 3) 3 clock cycles. The system is configured to exploit both LLP and ILP. Results are normalized to ST-io2.

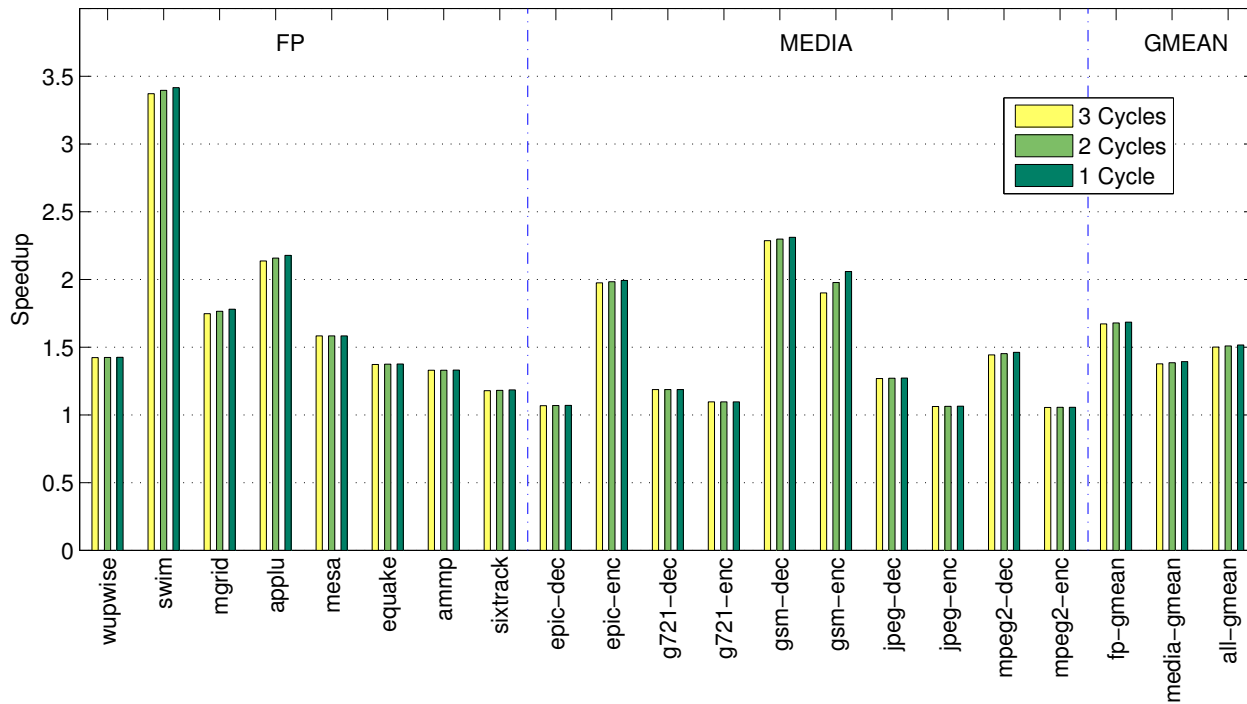


Figure 7.13: This figure shows the speedup of Tracy₃₄-io2 when the spill space has access latency of 1) 3 clock cycles, 2) 2 clock cycles, and 3) 1 clock cycle. The system is configured to exploit both LLP and ILP. Results are normalized to ST-io2.

example, *wupwise* has an average trace length of 4,196 instructions, but the performance of 8-way parallelization is actually worse than the speedup of 4-way parallelization. Lacking enough distant parallelism, over-parallelization using pre-inserted synchronizations actually hurts parallel performance. On the contrary, *applu* has an average trace length of 4,251 instructions, but the speedup achieved by 8-way parallelization is 1.80x and 1.25x better than the speedup achieved by 2-way and 4-way parallelization, respectively. The best performance scalability is observed in benchmarks that contain a large amount of LLP, which is not surprising because the same number of loop iterations can be distributed to a larger number of cores. The most obvious example is *swim*, whose parallel performance is almost increased proportionally to the increase of parallelization width. Thus, it is important to dynamically identify the “saturation point” for each program and only decomposes it into enough threads to exploit the available parallelism.

Figure 7.11 shows the performance of Tracy by varying the access latency of the synchronization array. Although six out of 18 benchmarks are actually slowed down when the access latency is eight clock cycles, an average speedup of 1.21x and 1.11x can still be achieved for floating point and media benchmarks, respectively. Thus, Tracy has the potential to be used on more commodity hardware that usually has relatively large synchronization overhead. On the contrary, such high synchronization overhead typically prevents most existing DIS-based DBP techniques from operating effectively. For benchmarks that contain a large amount of LLP, the access latency has the smallest impact on the program performance. For example, with eight-cycle access latency, the speedup of *swim* and *mgrid* is still 3.29x and 1.57x, respectively, which is very close to the speedup (3.41x and 1.77x, respectively) when the access latency is only one clock cycle.

Figure 7.12 shows the performance of Tracy by varying the transfer delay of the backbone bus. When it is increased to 12 clock cycles (four clock cycles per segment), an average speedup of 1.39x can still be achieved, which is only 1.08x worse than the speedup with three-cycle transfer delay (one clock cycle per segment). Thus, Tracy has the potential to master slave clusters that are placed distantly from one another on the single chip. For each benchmark, the sensitivity of its performance to the transfer delay is highly dependent on the amount of live-in and -out registers and cache lines. For example, 10.26% of the total execution time of *sixtrack* is used to transfer program state, so that the application is actually slowed down when the transfer delay is 12 clock cycles.

Figure 7.13 shows the performance of Tracy by varying the access latency of the spill space. For benchmarks that have relatively short traces, increasing the access latency to three clock cycles does not have any impacts since very few register spills are actually generated. In fact, the longer the trace is and the more distant parallelism exists, the more registers need to be spilled because aggressive

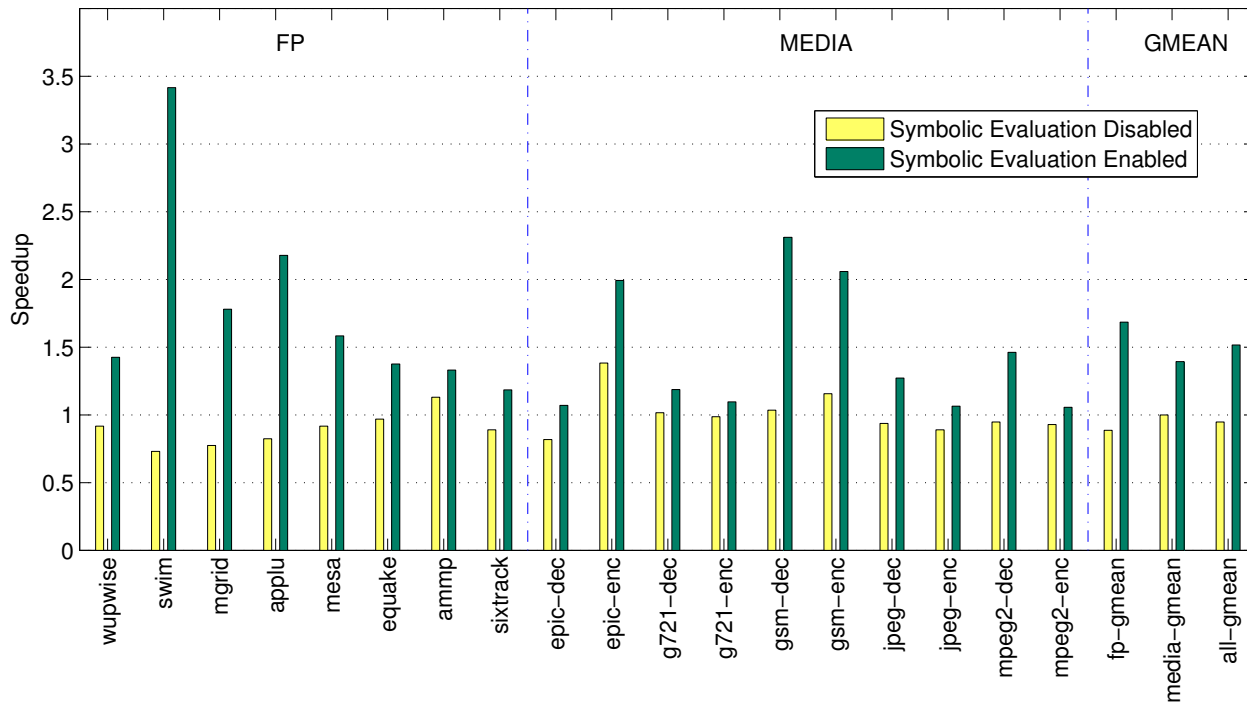


Figure 7.14: This figure shows the speedup of Tracy₃₄⁴-io2 when 1) symbolic evaluation is disabled, and 2) symbolic evaluation is enabled. The system is configured to exploit both LLP and ILP. Results are normalized to ST-io2.

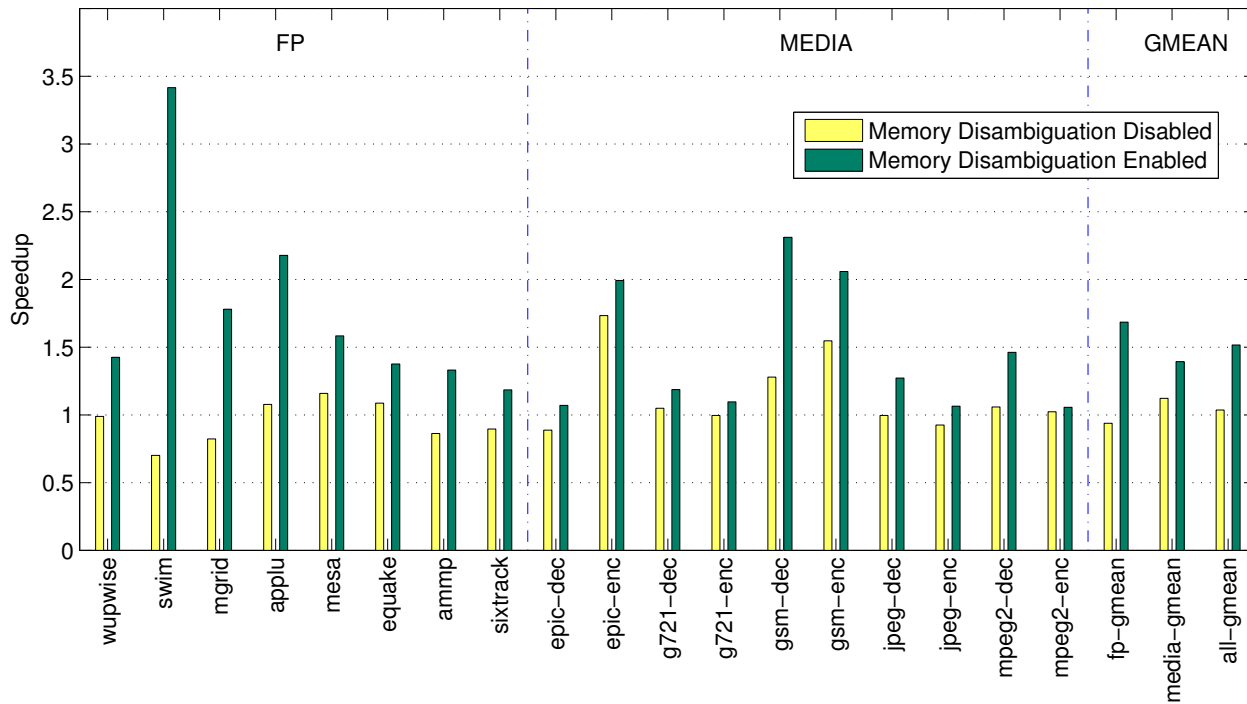


Figure 7.15: This figure shows the speedup of Tracy₃₄-io2 when 1) memory disambiguation is disabled, and 2) memory disambiguation is enabled. The system is configured to exploit both LLP and ILP. Results are normalized to ST-io2.

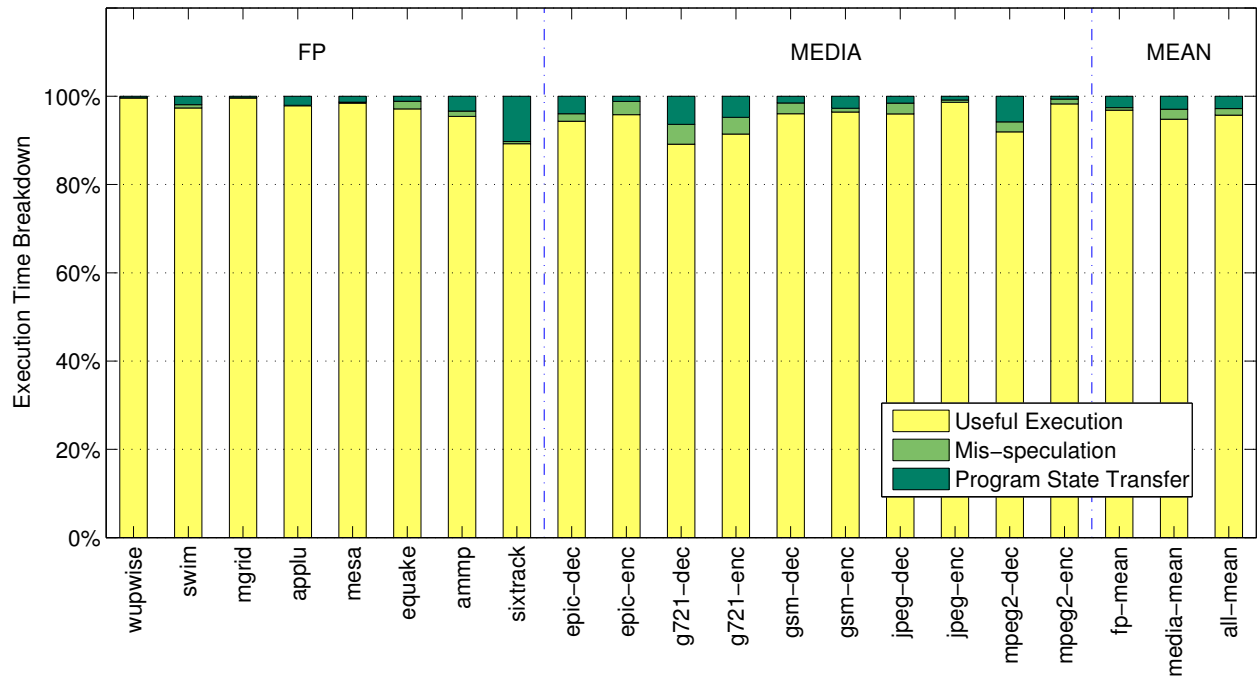


Figure 7.16: This figure shows the execution time overhead of Tracy₃₄⁴-io2 caused by mis-speculation and program state transfer. The system is configured to exploit both LLP and ILP. Useful execution includes the time spent on both sequential execution and parallel execution that is speculated correctly.

instruction scheduling has more chances to extend the live range of these registers. For example, the speedup of *applu* drops from 2.17x to 2.13x when the access latency is increased from one clock cycle to three clock cycles. This small performance difference, however, does not justify the removal of the spill space because when stored in the normal L1 data cache, these spilled registers have the potential to be polluted by other data, causing much larger overhead that is needed to access the L2 cache or even main memory.

7.5.5 Isolating Overheads and Benefits

In Section 6.4, we conclude that optimization alone is generally not enough to speed up applications. Both symbolic evaluation and memory disambiguation, however, eliminate unnecessary data dependencies (both register and memory) or increase data dependency lengths, preparing the code for future parallelization. As illustrated in Figures 7.14 and 7.15, the average speedup is dramatically reduced to 0.94x and 1.03x when symbolic evaluation and memory disambiguation is disabled, respectively. For many benchmarks, the non-optimized traces only expose very limited parallelism opportunities, and the corresponding speedup can hardly outweigh the overhead caused by mis-speculation and program state transfer. In several cases, Tracy makes incorrect decisions and over-parallelizes traces, which may dramatically slow down the application. For example, when no code transformation is performed, the slowdown of *swim* and *mgrid* is only 0.94x and 0.94x, respectively. When either symbolic evaluation or memory disambiguation is disabled, the slowdown of *swim* drops below 0.70x and the slowdown of *mgrid* drops below 0.77x.

It is interesting that symbolic evaluation has a larger impact on the parallel performance than memory disambiguation. In fact, symbolic evaluation is an important preparation to achieve effective memory disambiguation. Tracy divides all memory references into groups with different base registers and disambiguates them at the group granularity. Because the number of monitored base registers is limited, it is extremely important to make as many memory reference instructions as possible to share the same base register. One functionality of symbolic evaluation on memory loads and stores is to replace the base register with another register that is defined earliest in the trace and can be used to calculate the same effective address, automatically fulfilling this task.

Figures 7.16 illustrates the execution time overhead caused by mis-speculation and program state transfer, respectively. Useful execution includes the time spent on both sequential execution and parallel execution that is speculated correctly. Both overheads are generally small. Averaged over all benchmarks, only 1.50% of total execution time is caused by mis-speculation and only 2.79% of

total execution time is caused by program state transfer. Media benchmarks waste relatively more clock cycles on mis-speculation due to the relatively high misprediction rate (7.58% on average). On the contrary, floating point benchmarks waste more clock cycles on program state transfer because longer traces (1,090 instructions on average) modify more registers and cache lines that need to be transferred back to the sequential execution.

7.6 Summary

In this chapter, we leverage traces as the unified representation of program execution to exploit both coarse- and fine-grained parallelism. For exploiting ILP, Tracy customizes the traditional list scheduling algorithm [36] to partition and schedule instructions among different cores. For exploiting LLP, Tracy performs two major code transformations, accumulator expansion and dependent code motion, to eliminate loop-carried dependencies or at least to increase the execution overlap of multiple threads so as to achieve better parallel performance.

Generally speaking, only exploiting ILP produces greater speedups than only exploiting LLP, which is not surprising because it can schedule instructions at finer granularity. While the ILP-only speedup is 1.31x, averaged over all benchmarks, the LLP-only speedup is only 1.18x. When enough LLP is available, however, much larger speedups can be achieved. For example, LLP can be extracted from 81.94% of dynamic instructions in *swim*, leading to the great speedup of 3.34x. Although speeding up all floating point and media benchmarks, Tracy can hardly increase the performance of integer benchmarks due to their short trace length and low dynamic execution coverage, which is necessary to maintain high speculation accuracy. On the other hand, Tracy does consume more energy due to multi-trace execution and inter-core synchronization. When the background “system leakage” is accounted, however, Tracy only consumes 0.92x and 1.18x energy for floating point and media benchmarks, respectively. The performance of Tracy is quite sensitive to the core type. When using OoO cores instead of IO cores, Tracy can speed up six out of eight floating point benchmarks and four out of ten media benchmarks.

For floating point benchmarks, Tracy can achieve an average speedup of 2.16x (4-way parallelization), 1.51x better than the speedup achieved by Core Fusion. When Tracy performs 8-way parallelization, the speedup is further increased to 2.44x. For integer benchmarks, however, Core Fusion can actually achieve an average speedup of 1.3x, greatly outperforming the performance of Tracy. Thus, as we have hypothesized, Tracy outperforms DIS-based DBP when long traces can be constructed, which expose more distant ILP or even LLP opportunities.

Although the average speedup achieved by Tracy is only 1.04x better than the speedup achieved by CFG-based DBP, Tracy can speed up all floating point benchmarks while CFG-based DBP fails to parallelize three out of eight applications at all. When source code is not available, the necessity of conservative analysis on the CFG makes it much harder to extract LLP and CFG-based DBP lacks the capability to exploit ILP instead. Thus, we believe that Tracy is a favorable alternative because general applicability is very important for the success of any DBP techniques.

Chapter 8

Conclusions and Future Work

This dissertation explores the novel idea of trace-based DBP, which provides a large instruction window without introducing spurious dependencies. We hypothesize that traces provide a generally good trade-off between code visibility and analysis accuracy for a wide variety of applications so as to achieve better parallel performance. Compared to DIS-based DBP, trace-based DBP can exploit more distant parallelism because the average length of traces is typically much larger than the size of the hardware instruction window. Compared to CFG-based DBP, trace-based DBP does not need to respect control and data dependencies that are not on the execution path which is actually taken. More importantly, while DIS-based DBP typically only exploits fine-grained parallelism and CFG-based DBP typically only exploits coarse-grained parallelism, traces can be used as a unified representation of program execution to seamlessly incorporate the exploitation of both coarse- and fine-grained parallelism.

We first conduct a limit study to identify the performance limits of trace-based DBP. Based on the idealized assumption that an unlimited number of cores are available, the next executed trace can be accurately predicted by simply executing in parallel all existing traces that begin with the next target address. Thus, the average trace length is 235 basic blocks for integer benchmarks and 4,565 basic blocks for floating point benchmarks. Furthermore, only an average of 0.02% basic blocks for integer benchmarks and 0.19% basic blocks for floating point benchmarks are not formed into traces. Assuming one clock cycle for inter-core synchronization and one clock cycle to execute each instruction, the average speedup over sequential execution is 9.36x and 22.34x for integer and floating point benchmarks, respectively. Thus, the limit study demonstrates that trace-based DBP is a very promising technology for further exploration.

We then develop Tracy, an innovative DBP framework which monitors a program at run time and dynamically identifies hot traces, parallelizes them, and caches them for later use so that the program can run in parallel every time a hot trace repeats. Tracy supports multi-trace execution and holistically balances among trace length, speculation accuracy, and coverage of dynamic instructions. Tracy also performs two major optimizations, symbolic evaluation and memory disambiguation, which not only produce speedups directly, but also reformat the code to be more amenable to parallelism. Furthermore, Tracy customizes off-the-shelf algorithms to make them suitable for parallelizing atomic traces. When the system is configured as Tracy₃₄⁴-io2, the average speedup is 1.68x and 1.39x for floating point and media benchmarks, respectively. However, Tracy can hardly speed up integer benchmarks due to their short trace length and low dynamic execution coverage, which is necessary to maintain high speculation accuracy.

8.1 Merits of the Dissertation

History has shown that the HPC community will use all kinds of parallelization techniques to exploit the latest advances in many-core architectures that start to dominate the microarchitecture market. However, there is no precedent to show that non-DBP techniques will be adopted for mainstream computing, and in fact there are many reasons to believe otherwise. Some companies will undoubtedly continue to produce sequential programs due to the high cost of porting existing software, updating tool chains, and re-training employees. Furthermore, many legacy programs will be difficult or impossible to update with non-DBP techniques because source code is not available. Finally, DBP is important for portability and forward compatibility of the program, as the range and diversity of many-core architectures grow. These and other factors may be important enough that non-DBP techniques are not adopted in the mainstream, despite the lack of viable alternatives. The quality of DBP that can be effectively provided may be the limiting factor on the impact of many-core architectures on mainstream computing.

For floating point benchmarks, Tracy can achieve an average speedup of 2.16x (4-way parallelization), 1.51x better than the speedup achieved by Core Fusion. When Tracy performs 8-way parallelization, the speedup is further increased to 2.44x. For integer benchmarks, however, Core Fusion can actually achieve an average speedup of 1.3x, greatly outperforming the performance of Tracy. Thus, as we have hypothesized, Tracy outperforms DIS-based DBP when long traces can be constructed, which expose more distant ILP or even LLP opportunities.

Although the average speedup achieved by Tracy is only 1.04x better than the speedup achieved by CFG-based DBP, Tracy can speed up all floating point benchmarks while CFG-based DBP fails to parallelize three out of eight applications at all. When source code is not available, the necessity of conservative analysis on the CFG makes it much harder to extract LLP and CFG-based DBP lacks the capability to exploit ILP instead. Thus, we believe that Tracy is a favorable alternative because general applicability is very important for the success of any DBP techniques.

The current performance of Tracy neither matches the performance upper bound achieved by the limit study nor is always better than the speedup of existing DIS- and CFG-based DBP techniques. However, it takes the first step to dynamically parallelize the binary executable without using either the raw DIS or the complete CFG. Thus, this dissertation is expected have a broad impact on future researchers that explore other representations of program execution for DBP purposes. As depicted in Figure 1.1, any point on the design spectrum between the two extremes (DIS and complete CFG) may be the one that achieves the optimal trade-off between code visibility and analysis accuracy. Furthermore, the trace construction, prediction, optimization, and parallelization algorithms can be readily adopted by other dynamic trace-based systems.

8.2 Future Work

This dissertation only explores the possibility of using traces to provide a generally good trade-off on the design spectrum to a wide variety of applications so as to achieve better parallel performance. However, our experimental results have demonstrated that in order to maintain high speculation accuracy, traces generated from integer applications are typically too short to produce any speedups even on IO cores. This problem starts to affect floating point and media applications when the IO cores are upgraded to more advanced OoO cores.

One potential solution to the above problem is to merge traces into *partial CFGs*, which typically represent the execution of smaller code structures (e.g., part of a function instead of the entire function) than *static CFGs* in order to maintain a practical number of unmerged control flows for aggressive parallelization. On the other hand, these partial CFGs represent much longer program execution than traces, exposing more parallelism opportunities. The challenge, however, is how to balance the number of execution paths that the partial CFG represents and speculation accuracy so as to achieve the optimal trade-off. For example, if the partial CFG comprises one iteration of the outermost loop of a program, it may be easy to decide that an execution path that accounts for only 1% of all iterations should always be incorporated into the partial CFG if it does not introduce any

more dependencies. However, it is a much harder decision if the execution path accounts for 10% of all iterations but introduces two more dependencies, which may or may not affect the ultimate schedule length. As traces, partial CFGs can act as the unified representation of program execution to exploit both coarse- and fine-grained parallelism. In the following sections, we will describe the future research of effectively parallelizing partial CFGs in detail.

8.2.1 Balancing and Integrating Coarse- and Fine-Grained Parallelism

In the ideal case where a loop has many iterations and very low likelihood of loop-carried dependencies, all available cores should be allocated to execute loop iterations. In practice, however, the number of iterations is usually limited so that not all resources of the chip can be effectively utilized. Three scenarios exaggerate this problem. First, multiple iterations usually have to be packed into a single tile and executed on the same core, so that each parallel thread is large enough to tolerate large communication latencies. Loop tiling is also dictated by data locality and cache capacity considerations. Second, when loop-carried dependencies are more likely, only a few iterations are independent, and succeeding iterations must wait for values from their predecessors, limiting the degree of useful LLP to the loop dependency distance. Third, imperfect speculation accuracy interferes with execution of many subsequent iterations. For example, speculation accuracy of 90% means that parallel loop execution has to be suspended once every ten speculations on average. This aborted tile is executed sequentially and then parallel loop execution can be continued. This again limits the degree of useful LLP to the expected number of consecutive iterations that can be speculated successfully without any intervention.

In the above cases, it may be optimal to launch only a small number of iterations, and achieve further parallelism by using multiple cores to exploit ILP within each iteration. In future work, we can study how to balance the coarse- and fine-grained parallelism. The major challenge is to make trade-offs among various interacting factors, including average number of iterations, iteration size, loop-carried dependencies, speculation accuracy, and the potential benefits of allocating another core to exploit either LLP and ILP. Based on the achieved insights, we can then explore various decision algorithms based on performance prediction and runtime auto-tuning, and can develop new algorithms to exploit ILP within loop tiles.

8.2.2 Resource Allocation

Because Tracy uses additional cores at run time, it poses challenges for resource allocation. The challenges are of two types. The first one only applies when an application does not have the entire machine reserved, and other applications are contending for the same resources. Overall throughput and energy efficiency should be optimized subject to QoS constraints. This will require support in the task scheduler. For example, when an application creates a new thread, it should be able to convey its expected performance benefit as a result of the extra thread, allowing the scheduler to allocate resources among tasks of the same priority in a way that maximizes the system's desired figure of merit. Tracy then adapts based on whether the extra resources are allocated. Cheating can be prevented by the scheduler by monitoring performance and/or some form of credit system. The expected performance benefit can be based on a performance model or from historical information. If history is allowed to be maintained, an application using Tracy can also have its binary, or some associated metadata, augmented with prior performance as a function of the number of cores, allowing the additional cores to be requested when the application is first launched. Furthermore, when QoS constraints are present, the scheduler can use performance hints to judge whether allocating more resources would help an application that is falling behind its QoS target. We can evaluate these concepts in a prototype scheduler.

The second challenge pertains to energy efficiency. Tracy will be more widely adopted if it can monitor its own energy efficiency and throttle speculation as needed to maximize the system's desired figure of merit. We can implement this adaptation, using heuristics based on speculation accuracy, as well as exploring the value of more detailed information from performance counters.

8.2.3 Portability and Robustness to Runtime Dynamics

As a dynamic approach, Tracy already adjusts its parallelization strategy according to the number and type of cores available. We can extend this parameterization to support various asymmetric architectures, cost of inter-core communication, cache sizes, and so forth, and, via simulation of various architectures, test the portability of Tracy. We can also test its ability to respond to runtime dynamics, such as core availability (e.g., some cores may be unavailable due to hardware faults, power constraints, or contention with other applications), workload profile, etc.

Tracy can further adapt to runtime dynamics by observing data access patterns. Prior research has raised several orthogonal solutions, including 1) assigning operations that access similar data to the same processor [131], 2) building separate function versions to be executed on different core

types in an asymmetric or heterogeneous processor [76], and 3) co-scheduling appropriate tasks to reduce resource contention [140, 141]. All these approaches have achieved great performance improvement by considering resource *diversity* and *availability* during program parallelization, which can be explored in tuning the performance and robustness of Tracy.

8.2.4 Scalability

In cases where sufficient LLP is present (with high speculation accuracy), it may be worthwhile to launch work across multiple sockets or even multiple machines in a cluster. However, even though sockets on the same board share memory, parallelizing across sockets will incur high costs on inter-socket data transfers. We can develop heuristics that determine, based on trace-launch overheads, data-locality considerations, and possibly performance prediction and auto-tuning, how many resources to use, and the optimal size of loop tiles or other tasks. For multi-machine cases, Tracy will need to generate the appropriate MPI code. Furthermore, unless the MPI standard is extended, speculation on remote machines will need to be managed purely by software. While feasible in principle, these additional overheads will be prohibitive in many cases. However, in the interests of allowing users to achieve maximum performance, we can characterize cases in which this is beneficial and analyze whether MPI, operating system, or hardware support could increase the viability of trace-based DBP across multiple machines.

8.2.5 Combining Tracy with Native Parallelization

Often only regions of code that are easy to reason about are manually parallelized. Tracy can still be useful in parallelizing the rest of the code, including libraries and so forth that are not visible to the programmer. However, Tracy must not interfere with the user's explicit parallelization. This can easily be achieved by limiting Tracy to code regions with only one active thread. However, this may be unnecessarily limiting. One challenge to support Tracy with more than one live thread is that each thread must be traced. We can explore techniques for Tracy in the presence of multiple threads, including rotating active threads through the master core (as one thread reaches steady state, another thread can start tracing), cost-benefit analysis of providing tracing capabilities in multiple cores, and limiting tracing to long-running traces. Another challenge is contention for space of the trace cache. We hypothesize that this again can be addressed by waiting for threads to reach steady state before tracing a new thread, because the working set of useful traces will be much smaller than the set of candidates constructed when a thread is first traced. The third challenge is

contention for cores. Explicit threads must of course be assigned cores, but some threads will benefit more from Tracy than others. We can explore the use of performance prediction as well as runtime training to identify the most useful traces.

8.2.6 Implementing Tracy on Contemporary Hardware

To evaluate the *potential* of Tracy, we have assumed the presence of hardware such as support for trace construction, inter-core synchronization, register spilling, and memory disambiguation. If Tracy is sufficiently promising, hardware vendors may explore adding these features, and one line of our investigation can seek to evaluate the cost/benefit of such hardware as well as how to minimize the need for hardware support. But in many cases, especially for coarse-grained parallelism or high degrees of ILP, Tracy may be able to provide useful benefits without hardware support. This would increase the likelihood of adoption for Tracy and provide more immediate societal impact, which may subsequently make it easier to justify adding hardware support. Thus, we can develop a version of Tracy that can run on contemporary hardware.

Bibliography

- [1] Phillip Gibbons. Theory: Asleep at the Switch to Many-Core. <http://www.umiacs.umd.edu/~vishkin/T&MC5-2009/PRESENTATIONS/Gibbons.ppt>, May 2009.
- [2] Leland Freeman. Recover Missing Source Code to Overcome "Leaky-Roof Syndrome". *Enterprise Systems Journal*, October 1997.
- [3] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, June 2007.
- [4] Rakesh Ranjan, Fernando Latorre, Pedro Marcuello, and Antonio Gonzalez. Fg-STP: Fine-Grain Single Thread Partitioning on Multicores. In *Proceedings of the International Symposium on High Performance Computer Architecture*, February 2011.
- [5] David Tarjan, Michael Boyer, and Kevin Skadron. Federation: Repurposing Scalar Cores for Out-of-Order Instruction Issue. In *Proceedings of the Design Automation Conference*, June 2008.
- [6] Matthew DeVuyst, Dean Tullsen, and Seon-Wook Kim. Runtime Parallelization of Legacy Code on A Transactional Memory System. In *Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers*, January 2011.
- [7] Ben Hertzberg and Kunle Olukotun. Runtime Automatic Speculative Parallelization. In *Proceedings of the the International Symposium on Code Generation and Optimization*, April 2011.
- [8] Kanemitsu Ootsu, Takashi Yokota, Takafumi Ono, and Takanobu Baba. Preliminary Evaluation of a Binary Translation System for Multithreaded Processors. In *Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, January 2002.
- [9] Cheng Wang, Youfeng Wu, Edson Borin, Shiliang Hu, Wei Liu, Dave Sager, Tin-Fook Ngai, and Jesse Fang. Dynamic Parallelization of Single-Threaded Binary Programs using Speculative Slicing. In *Proceedings of the International Conference on Supercomputing*, June 2009.
- [10] Efe Yardimci and Michael Franz. Dynamic Parallelization and Mapping of Binary Executables on Hierarchical Platforms. In *Proceedings of the International Conference on Computing Frontiers*, May 2006.
- [11] Michael Gschwind. The Cell Broadband Engine: Exploiting Multiple Levels of Parallelism in A Chip Multiprocessor. *International Journal of Parallel Programming*, 35(3), June 2007.
- [12] M. Aater Suleman, Onur Mutlu, Moinuddin Qureshi, and Yale Patt. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2009.

- [13] Brian Fahs, Aqeel Maheesri, Francesco Spadini, Sanjay Patel, and Steven Lumetta. The Performance Potential of Trace-based Dynamic Optimization. Technical Report UILU-ENG-04-2208, University of Illinois at Urbana-Champaign, November 2004.
- [14] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2000.
- [15] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, March 2003.
- [16] Richard Hank, Scott Mahlke, Roger Bringmann, John Gyllenhaal, and Wen-Mei Hwu. Superblock Formation Using Static Program Analysis. In *Proceedings of the International Symposium on Microarchitecture*, December 1993.
- [17] Scott Mahlke, David Lin, William Chen, Richard Hank, and Roger Bringmann. Effective Compiler Support for Predicated Execution using the Hyperblock. In *Proceedings of the International Symposium on Microarchitecture*, November 1992.
- [18] Matthew Merten, Andrew Trick, Erik Nystrom, Ronald Barnes, and Wen-Mei Hwu. A Hardware Mechanism for Dynamic Extraction and Relay of Program Hot Spots. In *Proceedings of the International Symposium on Computer Architecture*, June 2000.
- [19] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen Keckler, and Charles Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the International Symposium on Computer Architecture*, June 2003.
- [20] Weifeng Zhang, Brad Calder, and Dean Tullsen. An Event-Driven Multithreaded Dynamic Optimization Framework. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2005.
- [21] Sanjay Patel and Steven Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, 50(6), June 2001.
- [22] Haitham Akkary and Michael Driscoll. A Dynamic Multithreading Processor. In *Proceedings of the International Symposium on Microarchitecture*, December 1998.
- [23] Pedro Marcuello and Antonio Gonzalez. Clustered Speculative Multithreaded Processors. In *Proceedings of the International Conference on Supercomputing*, June 1999.
- [24] Pedro Marcuello, Antonio Gonzalez, and Jordi Tubella. Speculative Multithreaded Processors. In *Proceedings of the International Conference on Supercomputing*, June 1998.
- [25] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and James Smith. Trace Processors. In *Proceedings of the International Symposium on Microarchitecture*, December 1997.
- [26] James Dehnert, Brian Grant, John Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *Proceedings of the International Symposium on Code Generation and Optimization*, March 2003.
- [27] Kemal Ebcioglu and Erik Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *Proceedings of the International Symposium on Computer Architecture*, June 1997.
- [28] Bolei Guo, Youfeng Wu, Cheng Wang, Matthew Bridges, Guilherme Ottoni, Neil Vachharajani, Jonathan Chang, and David August. Selective Runtime Memory Disambiguation in A Dynamic Binary Translator. In *Proceedings of the International Conference on Compiler Construction*, March 2006.

- [29] Gary Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the Symposium on Principles of Programming Languages*, October 1973.
- [30] Rastisalv Bodik and Sadun Anik. Path-Sensitive Value-Flow Analysis. In *Proceedings of the Symposium on Principles of Programming Languages*, January 1998.
- [31] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2002.
- [32] Brian Fahs, Satarupa Bose, Matthew Crum, Brian Slechta, Francesco Spadini, Tony Tung, Sanjay Patel, and Steven Lumetta. Performance Characterization of a Microarchitectural Framework for Dynamic Optimization. In *Proceedings of the International Symposium on Microarchitecture*, December 2001.
- [33] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on A Raw Machine. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
- [34] Walter Lee, Diego Puppini, Shane Swenson, and Saman Amarasinghe. Convergent Scheduling. In *Proceedings of the International Symposium on Microarchitecture*, November 2002.
- [35] Hongtao Zhong, Steven Lieberman, and Scott Mahlke. Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications. In *Proceedings of the International Symposium on High Performance Computer Architecture*, February 2007.
- [36] Keith Cooper, Philip Schielke, and Devika Subramanian. An Experimental Evaluation of List Scheduling. Technical Report CS-98-326, Rice University, September 1998.
- [37] John Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD Dissertation, Massachusetts Institute of Technology, February 1985.
- [38] Joseph Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30(7), July 1981.
- [39] Wen-Mei Hwu, Scott Mahlke, William Chen, Pohua Chang, Nancy Warter, Roger Bringmann, Roland Ouellette, Richard Hank, Tokuzo Kiyohara, Grant Haab, John Holm, and Daniel Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1), January 1993.
- [40] Eric Rotenberg, Steve Bennett, and James Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the International Symposium on Microarchitecture*, December 1996.
- [41] ChiKeung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapareddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2005.
- [42] Jing Yang, Shukang Zhou, and Mary Lou Soffa. Dimension: An Instrumentation Tool for Virtual Execution Environments. In *Proceedings of the International Conference on Virtual Execution Environments*, June 2006.
- [43] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack Davidson, David Evans, John Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and Practical Defense Against Code-Injection Attacks. In *Proceedings of the International Conference on Virtual Execution Environments*, June 2006.

- [44] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the USENIX Security Symposium*, August 2002.
- [45] Jason Mars and Mary Lou Soffa. Mats: MultiCore Adaptive Trace Selection. In *Proceedings of the Workshop on Software Tools for MultiCore Systems*, April 2008.
- [46] Sanjay Patel, Tony Tung, Satarupa Bose, and Matthew Crum. Increasing the Size of Atomic Instruction Blocks using Control Flow Assertions. In *Proceedings of the International Symposium on Microarchitecture*, December 2000.
- [47] Kevin Skadron. *Characterizing and Removing Branch Mispredictions*. PhD Dissertation, Princeton University, June 1999.
- [48] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, October 2000.
- [49] Bradford Chamberlain, David Callahan, and Hans Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3), August 2007.
- [50] Niklas Gustafsson. Axum Language Overview. <http://download.microsoft.com/download/B/D/5/BD51FFB2-C777-43B0-AC24-BDE3C88E231F/Axum%20Language%20Spec.pdf>, June 2009.
- [51] Antal Buss, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy Amato, and Lawrence Rauchwerger. STAPL: Standard Template Adaptive Parallel Library. In *Proceedings of the Haifa Experimental Systems Conference*, May 2010.
- [52] Monk-Ping Leong, Chi-Chiu Cheung, Chin-Wang Cheung, Polly Wan, Ivan Leung, Winnie Yeung, Wing-Seung Yuen, Kenneth Chow, Kwong-Sak Leung, and Philip Leong. CPE: A Parallel Library for Financial Engineering Applications. *Computer*, 38(10), October 2005.
- [53] Nuje Rucciuti. Y2K Cost Estimate Cut by \$2 Billion. http://news.cnet.com/Y2K-cost-estimate-cut-by-2-billion/2100-1091_3-235131.html, December 1999.
- [54] Jung Ahn, William Dally, Brucek Khailany, Ujval Kapasi, and Abhishek Das. Evaluating the Imagine Stream Architecture. In *Proceedings of the International Symposium on Computer Architecture*, June 2004.
- [55] Farhana Aleen and Nathan Clark. Commutativity Analysis for Software Parallelization: Letting Program Transformations See the Big Picture. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2009.
- [56] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers, October 2001.
- [57] Guilherme Ottoni and David August. Global Multi-Threaded Instruction Scheduling. In *Proceedings of the International Symposium on Microarchitecture*, December 2007.
- [58] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the International Symposium on Microarchitecture*, November 2005.
- [59] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew Bridges, and David August. Parallel-Stage Decoupled Software Pipelining. In *Proceedings of the International Symposium on Code Generation and Optimization*, April 2008.

- [60] Shane Ryoo, SainZee Ueng, Christopher Rodrigues, Robert Kidd, Matthew Frank, and Wenmei Hwu. Automatic Discovery of Coarse-Grained Parallelism in Media Applications. *Transactions on High Performance Embedded Architectures and Compilers*, 1(1), January 2007.
- [61] Gurindar Sohi, Scott Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the International Symposium on Computer Architecture*, June 1995.
- [62] Anasua Bhowmik and Manoj Franklin. A General Compiler Framework for Speculative Multithreading. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, August 2002.
- [63] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software Behavior Oriented Parallelization. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2007.
- [64] Troy Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-Cut Program Decomposition for Thread-Level Speculation. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2004.
- [65] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: A TLS Compiler that Exploits Program Structure. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, March 2006.
- [66] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen Keckler, and Charles Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the International Symposium on Computer Architecture*, June 2003.
- [67] Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd Mowry. A scalable approach to thread-level speculation. In *Proceedings of the International Symposium on Computer Architecture*, June, 2000.
- [68] Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd Mowry. Improving Value Communication for Thread-Level Speculation. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, February 2002.
- [69] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. Copy Or Discard Execution Model For Speculative Parallelization On Multicores. In *Proceedings of the International Symposium on Microarchitecture*, November 2008.
- [70] Neil Vachharaajani, Ram Rangan, Easwaran Raman, Matthew Bridges, Guilherme Ottoni, and David August. Speculative Decoupled Software Pipelining. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2007.
- [71] Anshuman Dasgupta. *Vizer: A Framework to Analyze and Vectorize Intel x86 Binaries*. Master of Science Thesis, Rice University, November 2002.
- [72] Aparna Kotha, Kapil Anand, Matthew Smithson, Greeshma Yellareddy, and Rajeev Barua. Automatic Parallelization in A Binary Rewriter. In *Proceedings of the International Symposium on Microarchitecture*, December 2010.
- [73] Benoit Pradelle, Alain Ketterlin, and Philippe Clauss. Polyhedral Parallelization of Binary Code. *ACM Transactions on Architecture and Code Optimization*, 8(4), January 2012.
- [74] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 1994.

- [75] Diego Llanos, David Orden, and Belen Palop. Just-In-Time Scheduling for Loop-based Speculative Parallelization. In *Proceedings of the Euromicro Conference on Parallel, Distributed and Network-Based Processing*, February 2008.
- [76] Michael Linderman, James Balfour, Teresa Meng, and William Dally. Embracing Heterogeneity – Parallel Programming for Changing Hardware. In *Proceedings of the Workshop on Hot Topics in Parallelism*, March 2009.
- [77] Nathan Clark. Why Should I Rewrite My Software When Dynamic Compilation Can Be Good Enough? In *Proceedings of the Workshop on Software Tools for Multi-Core Systems*, April 2008.
- [78] David Penry. You Can’t Parallelize Just Once: Managing Manycore Diversity. In *Proceedings of the Workshop on Manycore Computing*, June 2007.
- [79] John Hennessy, David Patterson, and David Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, September 2006.
- [80] Michael Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Companies, June 2003.
- [81] Borys Bradel and Tarek Abdelrahman. Automatic Trace-Based Parallelization of Java Programs. In *Proceedings of the International Conference on Parallel Processing*, September 2007.
- [82] Borys Bradel and Tarek Abdelrahman. The Potential of Trace-Level Parallelism in Java Programs. In *Proceedings of the International Symposium on Principles and Practice of Programming in Java*, September 2007.
- [83] Borys Bradel and Tarek Abdelrahman. The Use of Hardware Transactional Memory for the Trace-Based Parallelization of Recursive Java Programs. In *Proceedings of the International Conference on Principles and Practice of Programming in Java*, September 2009.
- [84] Raymond Buse and Westley Weimer. The Road Not Taken: Estimating Path Execution Frequency Statically. In *Proceedings of the International Conference on Software Engineering*, May 2009.
- [85] James Larus. Whole Program Paths. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 1999.
- [86] Xiangyu Zhang and Rajiv Gupta. Whole Program Traces. In *Proceedings of the International Symposium on Microarchitecture*, December 2004.
- [87] Qin Zhao, Ioana Cutcutache, and WengFai Wong. Pipa: Pipelined Profiling and Analysis on Multi-Core Systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, April 2008.
- [88] Martin Burtcher. VPC3: A Fast and Effective Trace-Compression Algorithm. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2004.
- [89] Jarek Nieplocha. Horizons in Extreme Scale Computing Software for Petascale Systems. <http://multiscale.emsl.pnl.gov/docs/presentations/wsu-08.ppt>, August 2008.
- [90] Samuel King, George Dunlap, and Peter Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference*, April 2005.

- [91] Min Xu, Rastislav Bodik, and Mark Hill. A “Flight Data Recorder” for Enabling Full-System Multiprocessor Deterministic Replay. In *Proceedings of the International Symposium on Computer Architecture*, June 2003.
- [92] Jesper Larsson and Alistair Moffat. Offline Dictionary-Based Compression. In *Proceedings of the Conference on Data Compression*, March 1999.
- [93] Quinn Jacobson, Eric Rotenberg, and James Smith. Path-Based Next Trace Prediction. In *Proceedings of the International Symposium on Microarchitecture*, December 1997.
- [94] YuKwong Kwok and Ishfaq Ahmad. Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. *Transactions on Parallel and Distributed Systems*, 7(5), May 1996.
- [95] ChiKeung Luk. Memory Disambiguation for General-Purpose Applications. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, November 1995.
- [96] Mikko Lipasti, Christopher Wilkerson, and John Shen. Value Locality and Load Value Prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.
- [97] Yiannakis Sazeides and James Smith. The Predictability of Data Values. In *Proceedings of the International Symposium on Microarchitecture*, December 1997.
- [98] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David August. Decoupled Software Pipelining with the Synchronization Array. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2004.
- [99] Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture*, February 2003.
- [100] Jason Hiser, Daniel Williams, Adrian Filipi, Jack Davidson, and Bruce Childers. Evaluating Fragment Construction Policies for SDT Systems. In *Proceedings of the International Conference on Virtual Execution Environments*, June 2006.
- [101] Jason Hiser, Daniel Williams, Wei Hu, Jason Mars, Bruce Childers, and Jack Davidson. Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, April 2008.
- [102] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip Gibbons, Todd Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *Proceedings of the International Symposium on Computer Architecture*, June 2008.
- [103] Brad Calder, Glenn Reinman, and Dean Tullsen. Selective Value Prediction. In *Proceedings of the International Symposium on Computer Architecture*, May 1999.
- [104] Pritpal Ahuja, Kevin Skadron, Margaret Martonosi, and Douglas Clark. Multipath Execution: Opportunities and Limits. In *Proceedings of the International Conference on Supercomputing*, July 1998.
- [105] Artur Klauser and Dirk Grunwald. Instruction Fetch Mechanisms for Multipath Execution Processors. In *Proceedings of the International Symposium on Microarchitecture*, December 1999.

- [106] Steven Wallace, Brad Calder, and Dean Tullsen. Threaded Multiple Path Execution. In *Proceedings of the International Symposium on Computer Architecture*, June 1998.
- [107] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the International Symposium on Microarchitecture*, December 2011.
- [108] Aniruddha Udupi, Naveen Muralimanohar, and Rajeev Balasubramonian. Towards Scalable, Energy-Efficient, Bus-Based On-Chip Networks. In *Proceedings of the International Symposium on High Performance Computer Architecture*, February 2010.
- [109] Sriram Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Priya Iyer, Arvind Singh, Tiju Jacob, Shailendra Jain, Sriram Venkataraman, Yatin Hoskote, and Nitin Borkar. An 80-Tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS. In *Proceedings of the International Solid State Circuits Conference*, February 2007.
- [110] Apala Guha, Kim Hazelwood, and Mary Soffa. Balancing Memory and Performance through Selective Flushing of Software Code Caches. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, October 2010.
- [111] Kim Hazelwood and Michael Smith. Generational Cache Management of Code Traces in Dynamic Optimization Systems. In *Proceedings of the International Symposium on Microarchitecture*, December 2003.
- [112] Alan Smith. Sequential Program Prefetching in Memory Hierarchies. *Computer*, 11(12), December 1978.
- [113] Vugranam Sreedhar, Guang Gao, and Yong-Fong Lee. Identifying Loops using DJ Graphs. *ACM Transactions on Programming Languages and Systems*, 18(6), November 1996.
- [114] Pablo Ortego and Paul Sack. SESC: SuperESCAlar Simulator. <http://iacoma.cs.uiuc.edu/~paulsack/sescdoc/>, December 2004.
- [115] Carl Ramey. TILE-Gx ManyCore Processor: HW Acceleration Interfaces and Mechanisms. In *Keynotes of the Symposium on High Performance Chips*, August 2011.
- [116] Robert Golla. Niagara2: A Highly Threaded Server-on-A-Chip. <http://www.opensparc.net/pubs/preszo/06/04-Sun-Golla.pdf>, October 2006.
- [117] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Transactions on Graphics*, 27(3), August 2008.
- [118] David Kanter. Cavium MIPSes Network and Security Processing. <http://www.realworldtech.com/page.cfm?ArticleID=RWT061206011113&p=2>, June 2006.
- [119] Marcus Yam. Intel's Knights Corner: 50+ Core 22nm Co-processor. <http://www.tomshardware.com/news/intel-knights-corner-mic-co-processor,14002.html>, November 2011.
- [120] Preston Briggs, Keith Cooper, and Linda Torczon. Coloring Register Pairs. *ACM Letters on Programming Languages and Systems*, 1(1), March 1992.
- [121] Gregory Chaitin. Register Allocation and Spilling via Graph Coloring. *SIGPLAN Notices*, 39(4), April 2004.

- [122] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions in Embedded Computing Systems*, 1(1), November 2002.
- [123] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the International Symposium on Hardware/Software Codesign*, May, 2002.
- [124] Hsien-Hsin Lee and Gary Tyson. Region-Based Caching: An Energy-Delay Efficient Memory Architecture for Embedded Processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, November 2000.
- [125] Rajiv Ravindran, Michael Chu, and Scott Mahlke. Compiler-Managed Partitioned Data Caches for Low Power. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2007.
- [126] Dirk Beyer, Adam Chlipala, Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. The Blast Query Language for Software Verification. In *Proceedings of the International Conference on Static Analysis*, August 2004.
- [127] Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Thread-Modular Counterexample-Guided Abstraction Refinement. In *Proceedings of the International Conference on Static Analysis*, September 2010.
- [128] Malay Ganai and Franjo Ivancic. Efficient Decision Procedure for Non-Linear Arithmetic Constraints using CORDIC. In *Proceedings of the Conference on Formal Methods in Computer Aided Design*, November 2009.
- [129] Sicun Gao, Malay Ganai, Franjo Ivancic, Aarti Gupta, Sriram Sankaranarayanan, and Edmund Clarke. Integrating ICP and LRA Solvers for Deciding Nonlinear Real Arithmetic Problems. In *Proceedings of the Conference on Formal Methods in Computer Aided Design*, October 2010.
- [130] Stefania Perri and Pasquale Corsonello. Fast Low-Cost Implementation of Single-Clock-Cycle Binary Comparator". *IEEE Transactions on Circuits and Systems*, 55(12), December 2008.
- [131] Michael Chu, Rajiv Ravindran, and Scott Mahlke. Data Access Partitioning for Fine-Grain Parallelism on Multicore Architectures. In *Proceedings of the International Symposium on Microarchitecture*, December 2007.
- [132] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke. Uncovering Hidden Loop Level Parallelism in Sequential Applications. In *Proceedings of the International Symposium on High Performance Computer Architecture*, February 2008.
- [133] Gang Chen. *Effective Instruction Scheduling with Limited Registers*. PhD Dissertation, Harvard University, March 2001.
- [134] Khaing Kyi and Weng-Fai Wong. Cooperative Instruction Scheduling with Linear Scan Register Allocation. In *Proceedings of the International Conference on High Performance Computing*, December 2005.
- [135] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of International Symposium on Computer Architecture*, June 2000.
- [136] Premkishore Shivakumar and Norman Jouppi. CACTI 3.0: An Integrated Cache Timing, Power and Area Model. Technical Report WRL-2001-2, HP Labs, August 2001.

- [137] Jose Renau, Karin Strauss, Luis Ceze, Wei Liu, Smruti Sarangi, James Tuck, and Josep Torrellas. Thread-Level Speculation on A CMP Can Be Energy Efficient. In *Proceedings of the International Conference on Supercomputing*, June 2005.
- [138] Jiang Lin, Hongzhong Zheng, Zhichun Zhu, Eugene Gorbatoov, Howard David, and Zhao Zhang. Software Thermal Management of DRAM Memory for Multicore Systems. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2008.
- [139] Urs Hoelzle and Bill Weihl. High-Efficiency Power Supplies for Home Computers and Servers. http://www.flowdas.com/blog/wp-content/uploads/2009/10/PSU_white_paper.pdf, September 2006.
- [140] Jason Mars, Lingjia Tang, and Mary Lou Soffa. Directly Characterizing Cross Core Interference Through Contention Synthesis. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers*, January 2011.
- [141] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2010.