Dynamic Trace Analysis with Zero-Suppressed BDDs

by

Graham David Price

B.S. University of Evansville, 2002

M.S. University of Colorado at Boulder, 2006

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Electrical, Computer, and Energy Engineering

2011

This thesis entitled:
Dynamic Trace Analysis with Zero-Suppressed BDDs
written by Graham David Price
has been approved for the Department of Electrical, Computer, and Energy Engineering

_____
Manish Vachharajani, Ph.D. (Chair)

_____
Fabio Somenzi, Ph.D.

_____
Bor-Yuh Evan Chang, Ph.D.

_____
Jeremy G. Siek, Ph.D.

_____
Tipp Moseley, Ph.D. (Outside Member)

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Price, Graham David (Ph.D., Electrical, Computer, and Energy Engineering)

Dynamic Trace Analysis with Zero-Suppressed BDDs

Thesis directed by Professor Manish Vachharajani, Ph.D. (Chair)

Instruction level parallelism (ILP) limitations have forced processor manufacturers to develop multi-core platforms with the expectation that programs will be able to exploit thread level parallelism (TLP). Multi-core programming shifts the burden of locating additional performance away from computer hardware to the software developers, who often attempt high-level redesigns focused on exposing thread level parallelism, as well as explore aggressive optimizations for sequential codes.

Precise dynamic analysis can provide useful guidance for program optimization efforts, including efforts to find and extract thread level parallelism. Unfortunately, finding regions of code amenable to further optimization efforts requires analyzing traces that can quickly grow in size. Analysis of large dynamic traces (e.g. one billion instructions or more) is often impractical for commodity hardware.

An ideal representation for dynamic trace data would provide compression. However, decompressing large software traces, even if decompressed data is never permanently stored, would make many analysis impractical. A better solution would allow analysis of the compressed data, without a costly decompression step. Prior works have developed trace compressors that generate an analyzable representation, but often limit the precision or scope of analyses.

Zero-suppressed binary decision diagram (ZDDs) exhibit many of the desired properties of an ideal trace representation. This thesis shows: (1) dynamic trace data may be represented by zero-suppressed binary decision diagrams (ZDDs); (2) ZDDs allow many analyses to scale; (3) encoding traces as ZDDs can be performed in a reasonable amount of time; and, (4) ZDD-based analyses, such as irrelevant instruction detection and potential coarse-grained thread level parallelism extraction, can reveal a number of performance opportunities in sequential programs.

Dedication

Acknowledgements

Contents

Chapter

Figures

Figure

Chapter 1

Introduction

Instruction level parallelism (ILP) limitations have forced processor manufacturers to develop multi-core platforms with the expectation that programs will be able to exploit thread level parallelism (TLP) [1–3]. This shift forces software engineers to try and improve the performance of their applications with high-level redesigns focused on exposing parallelism, as well as explore aggressive optimizations for sequential codes [39, 71]. Extracting TLP by manual high-level software redesign is often difficult [40]. Instruction level dynamic program analysis can provide useful guidance for program optimization, including efforts to find and extract thread level parallelism. However, optimizations often exist within large dynamic traces [44], and finding opportunities for performance can require the analysis of gigabytes of trace data. This thesis shows that: (1) Zero-Suppressed Binary Decision Diagrams (ZDDs) enables many analyses to scale; (2) ZDD creation is practical for traces of a billion instructions for a variety of benchmarks; and, (3) ZDD-based analysis, such as irrelevant instruction detection and potential coarse-grained thread level parallelism extraction, can reveal a number of performance opportunities that exist in sequential programs.

1.1 Dynamic Trace Compression

Dynamic trace analysis has been used in prior work for performance tuning and hardware debugging [102]. Unfortunately, trace files can easily grow to terabytes in size depending on the information collected and the duration of traced execution. Large dynamic trace sizes (e.g. 1 billion instructions or more) can make analysis and visualization impractical.

Researchers have developed streaming compression algorithms (e.g., Burtscher *et. al.* [18]) that can

compress these traces by a factor of 10 or more. Unfortunately, these techniques do not speed trace analysis. Compression techniques force analyzers to stream a decompressed version of the trace through the analysis engine. Thus, analyses have complexity that depends on the *decompressed* trace size, even though the decompressed trace is never stored on disk. With large traces, this time-consuming process prohibits certain global analyses and interactive tools. Examples of prohibitively expensive operations include memory-data liveness visualization, hot code visualization, trace slicing [105], and interactive visualization of thread level parallelism.

Ideally, a compressed trace format should allow analyses to operate directly on the compressed representation with complexity that is a function of *compressed* trace size. Then, if large portions of compressed traces fit in memory, global analyses and interactive visualization become possible. Larus *et. al.* propose such a technique for *whole program path* analysis [58]. The technique uses the SEQUITUR compression method and works well for finding sequence matches in program execution. However, *whole program path* analysis does not permit direct application of data-centric analyses (e.g., trace slicing) that are of interest to system designers and programmers. Recent work on stride compression techniques addresses this issue by forming hierarchies based on accessed memory regions [50], but is limited to analyses based on loop-level dependence.

This thesis explores reduced, ordered, binary decision diagram (ROBDDs) [17], originally developed for hardware verification, as a trace representation for dynamic program analysis. BDDs can provide compression for large sets of data whose size would otherwise make analysis intractable. For example, BDDs in hardware verification and validation allow equivalence checking of circuits with many states in constant time [14]. In program analysis, BDDs have been used to store program contexts for each object in a program analysis lattice object [99]. When BDDs are used for the analysis of large program traces [75, 78, 105], the size of dynamic program traces can be reduced by up to 60x when encoded as a BDD [75]. Further, this compressed representation can be analyzed without decompression, with algorithmic complexity that is a function of the compressed size [75]. Thus, trace-encoded BDDs provide a solution to the dynamic trace size issue by representing trace information in a compressed, yet analyzable, structure.

1.2 Dynamic Trace Analysis at Scale

Encoding large traces as BDDs can be time consuming, requiring hours to days to complete [77]. This in turn makes tools that use BDD-based representations less effective than otherwise possible. Prior applications of BDDs depend on three methods to mitigate BDD creation time: (1) search for a variable order that allows for fast BDD creation, (2) tune the tables and caching systems used in many BDD packages, and (3) encode an abstraction of the original data set. This thesis discusses zero-suppressed BDDs (ZDDs) as an alternative to BDDs in order to reduce creation time for large traces. Prior work has shown that ZDDs can reduce the final BDD size for sparse data and context data used during static program analysis [43, 62], though this work has not applied ZDDs to compressing dynamic trace data. This thesis shows that, without data loss, ZDD-based SPEC INT 2000 benchmark traces are 25% smaller than BDD-based traces.

Traces from a variety of applications need be analyzed to demonstrate the efficacy of ZDD-based dynamic trace analysis. Reducing trace creation time by 25%, which intuitively should correspond to the 25% reduction in representation size, is beneficial, but simply not enough to allow analysis of billions of instructions from a variety of benchmarks. Initial tests of ZDD creation time proved to be far worse; the 25% reduction in representation size did not translate to 25% reduction in creation time. ZDDs creation time was, for some benchmarks, $3\times$ slower. Further investigation revealed a modification to the caching mechanism will remove this penalty in most cases. In fact, ZDD-based trace compression algorithms have a smaller working set size making tuning possible for large traces, which results in a creation time that can be $9\times$ faster than BDD creation time for the same benchmark. A detailed discussion of ZDD-based trace compression can be found in Chapter 4.

1.3 Opportunities for Optimization

Reducing ZDD creation time is crucial to explore opportunities for optimization in a wide range of program traces. Chapter 7 explores potential coarse-grained thread level parallelism (TLP) that exists in sequential applications. Chapter 6 demonstrates the use of ZDD-encoded traces to locate irrelevant instruction chains.

Irrelevant Instruction Elimination

Irrelevant component elimination has been used in prior work to simplify abstractions for static analysis [23]. This thesis uses an irrelevant component elimination with ZDD-encoded precise dynamic instruction dependencies. This thesis will show that ZDD-based irrelevant instruction dependency elimination can locate instructions that fail to the following criteria: (1) the instruction dependence chain should reach the end of the program trace; or, (2) the dependence chain should produce an output through a Linux system call.

ZDD-based irrelevant instruction dependency elimination is designed to iterate irrelevant code calculation until convergence. However, irrelevant instruction dependency elimination can take days to complete for traces with long dynamic dependency chains. Empirical data presented in this thesis shows that, for all benchmarks in SPEC 2006 INT, the irrelevant instruction elimination algorithm reaches a steady state. In this state, the number of instruction dependencies removed per slice iteration oscillates, but will not monotonically decrease until the analysis converges. Thus, it is possible to approximate the number of instructions removed by iterating irrelevant instruction elimination until oscillation is detected.

ZDD-based irrelevant instruction dependency elimination may also be used to filter irrelevant points from program visualizations. However, irrelevant instruction dependency elimination may also be used as a technique for code optimization or compiler evaluation. Chapter 6 contains a survey of irrelevant code removal from the SPEC 2006 INT benchmarks using both *-O0* and *-O2* optimization levels in the *gcc* compiler.

Hot-Code Visualization

In addition to the optimization analyzes presented in this thesis, which include coarse-grained thread level parallel region location and irrelevant instruction elimination, a hot-code visualization algorithm was created to focus optimization efforts on the most frequently executed regions of code.

Hot-code analysis captures static instruction execution frequency and counts the frequency of execution. The static hot code information is combined with a mapping from $dynamic\ instruction \rightarrow static\ instructions$ to create a new relation from each dynamic instruction to hot-code value. The resulting visualizations can be found in Appendix C and Appendix D.

1.4 Contributions

Traces from a variety of applications need be analyzed to demonstrate the efficacy of DD-based dynamic trace analysis. The results in this thesis show that ZDD-based trace compression results in 25% smaller representation compared to BDD-based traces. Further, ZDDs have a smaller working set, thus the ZDD creation package can tuned to cache the working set of the trace-ZDD during creation. This reduces the number of garbage collection operations and removal of useful dead nodes. This reduces DD creation by up to $9\times$.

Hot code analysis can tell developers where to focus optimization and parallelization efforts. In addition to the optimization analyzes presented in this thesis, which include coarse-grained thread level parallel region location and irrelevant instruction elimination, a hot-code visualization algorithm was created to focus optimization efforts on the most frequently executed regions of code.

The results from a survey of irrelevant instructions in the SPEC 2006 INT benchmark shows that over 50% of instruction dependencies do not produce a value and do not reach the end of the program trace. It is possible for an instruction to be relevant to program execution but not meet the specified requirements. Therefore, this thesis also presents results comparing the irrelevant dependence counts from both the *-O0* and *-O2* compiler settings. The irrelevant dependency count from the *-O0*, or non-optimized, compiler setting provides a worst-case value to normalize further comparison operations. Furthermore, this technique can test the effectiveness of static compiler optimizations, as well as locate potential irrelevant instruction streams.

This thesis explores potential coarse grain TLP that may be exploitable in conjunction with TLS and ILP techniques. In particular, the thesis examines the SPEC INT 2006 benchmark suite, looking for parallelism with a granularity of thousands of dynamic instructions, and is not restricted to loop-level TLP. The survey presented in Chapter 7 found, on average, 7% of instructions may be extracted as course-grained parallelism, and for the benchmark 445.gobmk, 44% of instructions may be extracted as coarse-grained TLP.

Potential Coarse-Grained Thread Level Parallelism

An effective, popular, and widely studied mechanism for automatically exploiting parallelism is to dynamically and speculatively thread groups of instructions [15, 20, 25, 64, 73, 79, 87, 90, 93, 97, 101]. Recent

advances in this thread level speculation (TLS) can parallelize execute over 90% of some codes [64]. However, TLS predictor accuracy limits TLS to fine-grained or loop-level TLP [15, 64, 79, 90, 98].

This thesis explores potential coarse grain TLP that may be exploitable in conjunction with TLS and ILP techniques. In particular, the thesis examines the SPEC INT 2006 benchmark suite, looking for parallelism with a granularity of thousands of dynamic instructions, and is not restricted to loop-level TLP. Coarse-grained TLP is located using the ParaMeter dynamic trace visualization tool [78]. This technique, which is discussed in Section 7.1, generates a visualization of program execution, called a DINxRDY (dynamic instruction number by ready time) plot. This plot visually shows potential coarse grain TLP as lines that overlap on the x-axis.

Potential parallel regions found within DINxRDY plots are further analyzed to expose dependence relationships. Inter-region dependence conflicts, discussed in Section 7.4 are found by dynamic dependence graph (DDG) slicing [4,5,30,52]. Slicing DDGs that contain a large number of instructions (e.g. billions) can take weeks [5,104]. To efficiently, and precisely, explore the dependency relationships between two regions of code this thesis extends the DDG *chop* to use the ZDD-compressed trace format. The dynamic chop [36, 54] is often faster than an intersection of a forward and reverse slice and requires no loss of precision.

The survey presented in Chapter 7 found, on average, 7% of instructions may be extracted as course-grained parallelism, and for the benchmark 445.gobmk, 44% of instructions may be extracted as coarse-grained TLP.

Finally, a summary of contributions, techniques, and results of this thesis may be found in Chapter 9.

Thus, this thesis presents the following contributions:

(1) A ZDD-based trace compression algorithm with tuning for large traces, resulting in a 25% reduction in BDD size and a $9\times$ reduction in trace compression time.

(2) A ZDD-based iterative analysis for locating irrelevant instruction dependencies

(3) A method to quickly explore coarse-grained parallelism in serial applications.

(4) A ZDD-based dependence graph chopping algorithm need for the aforementioned method.

(5) A survey of coarse-grained thread level parallelism in the SPEC INT 2006 benchmarks that shows up to 44% of instructions may be extractable as coarse grain TLP.

1.5 Hypothesis

ZDD-based dynamic trace analysis can identify hot-code paths, irrelevant instructions, and potential coarse-grained thread level parallelism in sequential codes, and thus create opportunities for program optimization.

Chapter 2

Parallel Computation Background

Parallel computation is a source for performance for both software and hardware in many modern computer systems [1–3]. Correct manual thread level parallelization of software can be difficult, may result in non-deterministic errors, and often the resulting parallel application is often harder to debug than a sequential application [29, 40, 57]. This thesis explores software tools that help a developer find and extract parallelism. Parallel computation and parallel programming would require volumes to discuss completely [40, 61]. This thesis requires an understanding of three attributes of parallel tasks: (1) location; (2) granularity; and, (3) conflicts. Prior works have researched both automatic and manual techniques for defining the location, granularity, and conflicts in potential parallel tasks. Following a discussion on these topics, this chapter will then explore static and dynamic techniques that can locate parallel tasks and task conflicts.

2.1 Automatic Parallelization Techniques

Automatic parallelization is a heavily research field [38, 41, 95]. Parallel task location and granularity are defined by instructions contained within a parallel task boundary [19]. Conflicts between parallel tasks often occur from read-after-write hazards; if a task alters a value read by a different parallel task can result in unexpected program execution. This section looks at static and dynamic techniques for automatically finding parallel task locations, boundaries, and conflicts.

```
1:    a = &(arrayB[4])

2:    for (i = 0; i < arrayB.size(); ++i)
3:    {
4:      b = funcA()
5:      c = arrayB[i]
6:      arrayB[b + c] = i
7:    }

8:    print(*a)
9:    print(b)
```

Figure 2.1: Parallel Pseudo Code Example

2.1.1 Automatic Parallelization by Static Analysis

The location of parallelism can be determined a statically before program execution by a compiler [16, 25]. The example code in Figure 2.1 can help explain how to find parallel tasks and task boundaries using only static program information. An illustration of the sequential execution of the source in Figure 2.1 can be seen in Figure 2.2. This code is in C/C++ style, and contains many pointer operations to demonstrate the limitations of some analysis methods. In line 1, the variable *a* is a integer pointer variable, and the arrayB contains integers. Thus, line 1 sets the value of *a* to a pointer to the fourth member of arrayB.

The next point of heap obfuscation occurs inside the control flow test in line 2. The function *size* is a member of the array type used in this pseudo code. This function determines the size of the array, which can be difficult to determine at run-time. Thus, a static analysis would likely need to be *context sensitive* and *interprocedural*. Context sensitivity allows the analysis to understand the difference, or similarities, between multiple occurrences of the same static code [28]. An interprocedural analysis allows the analysis to go into, and out from, a function call during analysis [8]. A context sensitive and interprocedural analysis would still need to determine the size of the array *arrayB* which was likely allocated using heap memory. In languages that allow heap allocation and references, the size and value of heap memory is difficult to determine at compile-time difficult [51, 66, 92].

The size of the array may require an expensive points-to analysis [86], or, if the size of the array varies

with a run-time input, than the size of the array could not be found by static analysis. Interprocedural static analysis, such as *points-to* [86] and *alias* [22] can find conflicts between instructions.

The final heap accesses in the pseudo code in Figure 2.1 occur in lines 5 and 6. The heap read in line *c = arrayB[i]* would a points-to analysis to disambiguate. The next line, *arrayB[b + c] = i*, is a write to a heap memory location. If a static analysis can find the exact location of the memory write, then it is possible to perform additional parallelization analyses. However, the heap write access location is the combination of a function return value and the value in another heap memory location. It is likely that a static analysis would need to classify this memory write location as all possible heap location values, or *top* using a lattice that also represents the empty set $\emptyset$ as *bottom* [24, 89]. Top is commonly written as $\top$, and bottom is represented by $\bot$. If line 6 in Figure 2.1 is found to be $\top$ by a static analysis, it will cause line 6 to conflict with all other heap memory writes, and add line 6 to the dependence chain of a heap memory read. Thus, line 6 would become difficult to parallelize using static auto-parallelization.

The example code also contains potential parallelism that is not bounded by a programming construct. For example, if the value of *b + c* in line 6, *arrayB[b + c] = i*, is less than four, than the instructions *a = &(arrayB[4])* followed by *print(*a)*, and the loop *For (i = 0; i ¡ arrayB.size(); ++i)* may be able to execute in parallel. This program execution is shown in the illustration in Figure 2.4.

Figure 2.3 contains an illustration of the execution of the pseudo-code in Figure 2.1 using medium-to-fine grained parallel tasks bounded by control flow constructs. This illustration assumes a static, likely points-to, analysis finds that line 6 does not alter the heap locations read by line 5. Even if this is not the case, and line 5 depends on a value written by line 6, it may still possible to create parallel software pipelines [32, 81].

Figure 2.1 demonstrates how the boundary of parallel tasks located by static analysis can be limited by control paths that are resolved at run-time [95]. Prior work has found that even state-of-the-art automatic parallelizing compilers are often unable to parallelize simple embarrassingly parallel loops written in C/C++ [50].

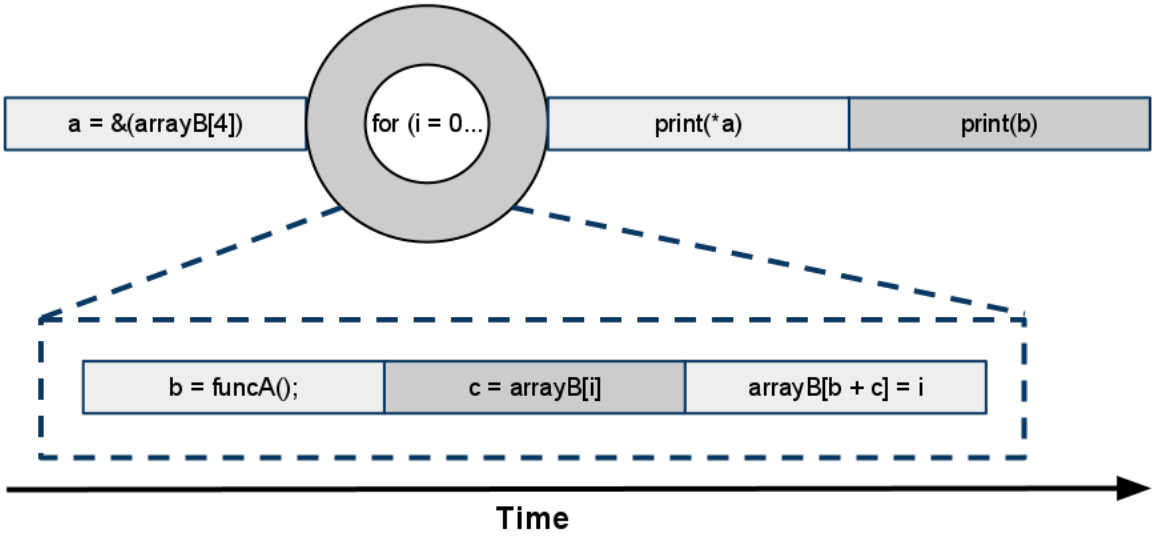


Figure 2.2: Sequential Execution of Program in Figure 2.1

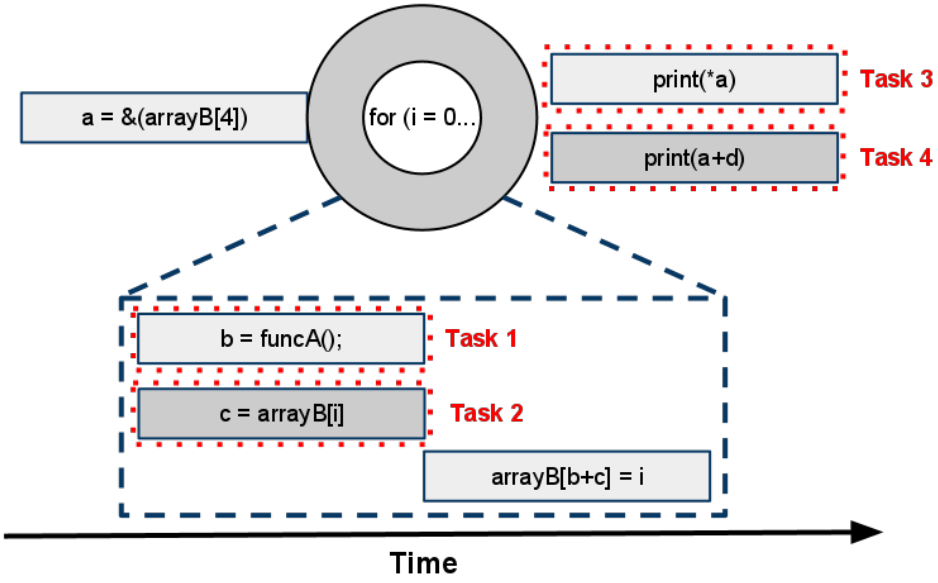


Figure 2.3: Static Fine-To-Medium Grained Parallel Tasks from Figure 2.1

2.1.2 Automatic Parallelization by Dynamic Analysis

Dynamic auto-parallelization techniques examine the run-time execution of a program, and thus can see how control flow dependencies are resolved. This illustration in Figure 2.4 show the pseudo-code in Figure 2.1 could be parallelized if a dynamic analysis were able to extract all potential parallel instructions. Note that the fine grained parallel tasks from Figure 2.3 are still executed as parallel tasks in Figure 2.4 spawned from Task 2.

An illustration of dynamic execution of the pseudo code in Figure 2.1 is shown in Figure 2.5. The *for* loop from the static code is shown partially unrolled in the the dynamic execution illustration.

Parallel tasks found by a dynamic analysis are only correct for the program input, or the set of inputs, that have been analyzed. Therefore, to ensure correct execution, thread-level parallelism found and executed dynamically is often executed speculatively. There are many works exploring thread-level speculative (TLS) techniques [20, 25, 69, 82, 87, 93, 97, 98, 101]. This thesis does not require a detailed understanding of transactions and speculation, but basic concepts are necessary. If two tasks are executed speculatively, then, for this work, it is assumed that the program executed in such a way that the final program state is identical to that of the sequentially executed code.

The illustration of the pseudo code in Figure 2.1 contains the dynamic code execution with TLS from Figure 2.5. The regions of this illustration contained by *Transaction 1* or *Transaction 2* may be rolled back by our hypothetical TLS system. Note that this illustration does not include nesting TLS; this is also true for many research TLS systems [74].

TLS often incorporate prediction to extract additional parallelism [48]. Prediction systems used by TLS can speculate values, control directions, and data dependencies. Figure 2.7 shows an illustration of a TLS system using control prediction to execute each iteration of the *for* loop in parallel. Value prediction in Figure 2.7 also allows the value of *i*, which is read by instructions inside the *for* loop, to be predicted and propagated.

The illustration in Figure 2.6 can help explain why TLS is often most effective when extracting threads with fewer than 1000 instructions [64]. TLS will often execute many iterations of If any instruction in *Task 2*

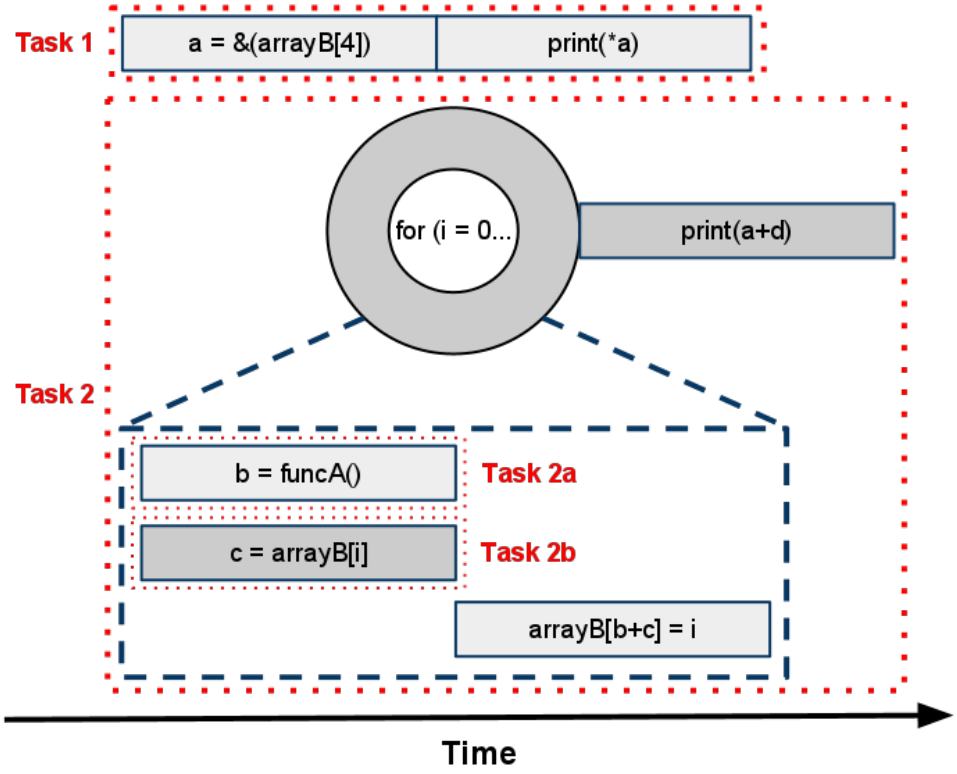


Figure 2.4: Static Fine-to-Coarse Grained Parallel Tasks from Figure 2.1

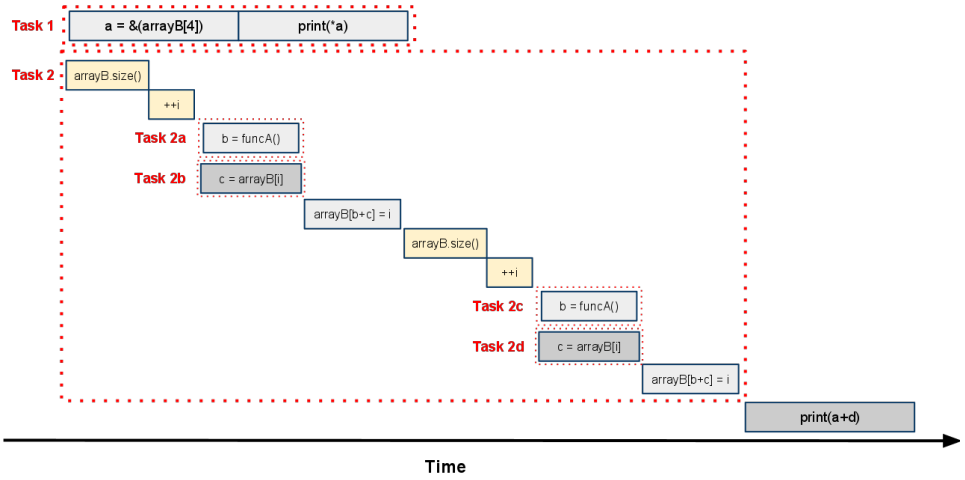


Figure 2.5: Dynamic Coarse Grained Parallel Tasks from Figure 2.1

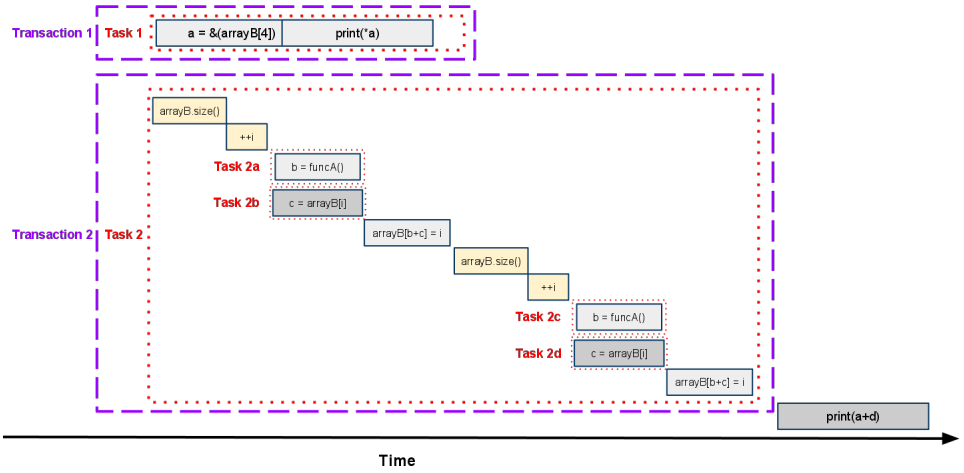


Figure 2.6: Dynamic Coarse Grained Parallel Tasks from Figure 2.1 with TLS

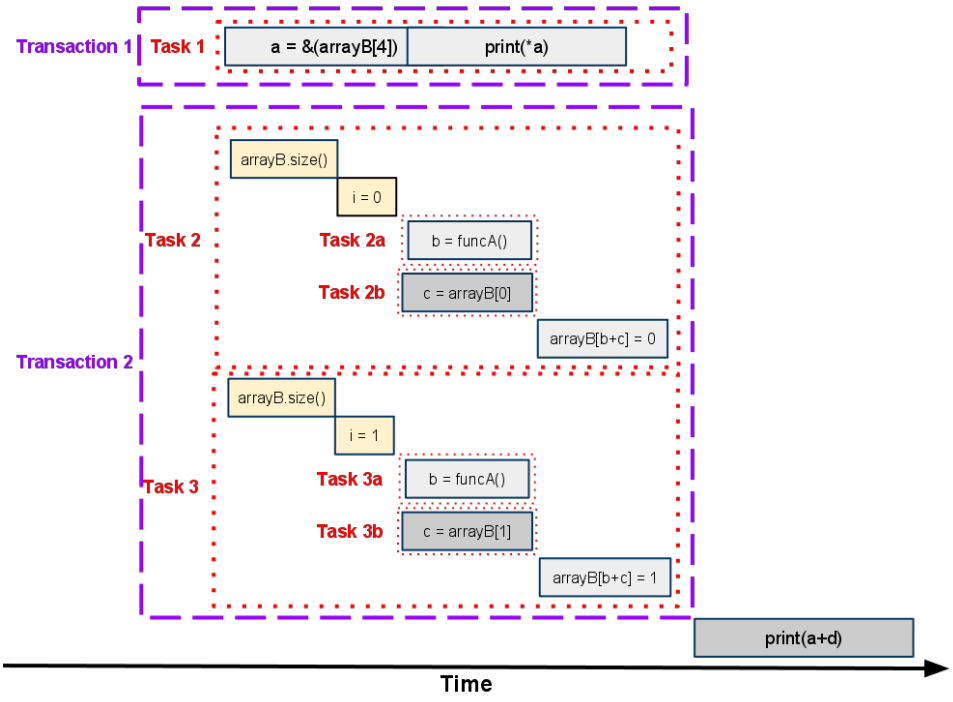


Figure 2.7: Dynamic Coarse Grained Parallel Tasks from Figure 2.1 with TLS and Prediction

Benchmark	TLS Perf
400.perlbench	6.47
401.bzip2	12.31
403.gcc	14.14
429.mcf	17.7
445.gobmk	12.78
456.hmmer	0.1
458.sjeng	16.1
462.libquantum	0.1
464.h264avc	3.34
471.ommnetpp	15.5
473.astar	40.67
483.xalancbmk	9.1

Figure 2.8: Performance from TLS [48]

conflict, then *Transaction 2* will likely roll back the entire transaction. Complex tasks may increase the risk of misprediction by including more predicted values, dependencies, and branch directions [64]. Work by Ravi *et. al.* looks at a technique based on transactional memory to expand TLS to coarse-grained threads, but is limited to scientific codes with regular access patterns [80]. TLS has been applied to C and Java objects that contain thousands of instructions, but work by Warg *et. al.* found limited gains were realized from objects with more than 100 instructions [98].

TLS can greatly improve program performance, regardless of the granularity limitations. For example, TLS performance gains from a TLS system that uses perform control, data dependence, and data value speculation [48], are shown in Figure 2.8.

2.2 Parallelization with Tools

Automatic parallelization is difficult, and most successes are limited to highly numeric applications [60, 80]. This section examines tools that can help a programmer find potential parallel tasks and find potential task conflicts. Static parallelization tools have had some success [49], but are ultimately limited in the presence of pointers and dynamic memory allocation [50]. This section explores tools that use run-time program information, much like TLS, for finding potential parallel tasks and task conflicts.

Commercial applications that use dynamic information to help find potential parallelism generally

keep the science behind these systems a secret [21, 94]. A limited number of tools have recently emerged from the research community, and are easier to explore for this section.

Alchemist is a dynamic profile based tool that locates potential parallelizable regions at run-time. Dependence conflicts are also located and presented to a user for manual parallelization [106]. The Alchemist tool restricts parallel tasks to regions bounded by artificial constructs, like *if-than-else* branches or loops.

Recent research into parallel extraction tools include SD3 [50], Parkour [45], and Kremlin [31]. The SD3 tool performs inter-region dependence analyses similar to those discussed in this work. The regions located by SD3 are defined stride-compression, and therefore are generally limited to locating potential parallelizable regions in loops. Many recent advances in parallel task boundary locations are also limited to parallelism bounded by explicit constructs [31, 45].

Most tools that use dynamic program information to aid parallel programming are restricted to parallel tasks that can be bounded by *if-than-else* blocks, *for* or *while* loops, or function calls [31, 45, 50, 106]. This limitation is likely in place for the following reasons: (1) it is unclear how to find task boundaries without constructs; (2) constructs help trace compression; and, (3) frequently executed code blocks are often contained inside loops.

Constructs Create Boundaries

Artificial constructs are an easy way to define a parallel task boundaries. If the user defined a task to be contained in a function call, than that parallel task should be easy for the user to identify. Using techniques prior to the work in this thesis, it is not clear how to define the boundaries of these parallel tasks [31, 45]. Further, some works include the "ease of thread extraction" in calculations for deciding if a task is worth parallelizing [31]. Unfortunately, there may be groups of instructions that could be a parallel task, but are not contained by a construct in the static code.

Constructs Help Compression

Dynamic program analyses often need some dynamic information to be visible for the life of the program execution. Dynamic trace information can grow very large (e.g. gigabytes), and requires some form of analyzable compression. A potential parallel task that is bounded by a static construct can be stored for later analysis by only storing the static information. Multiple occurrences of the same construction may

require storing additional context information, or multiple occurrences may be ignored [31, 45, 50]. Storing only instances of the static code requires much less space than a trace of the dynamic execution, but at the cost of a loss of precision and flexibility. For example, if the tool stores only static data than it would be difficult to know how many times that region of code was executed. The execution count can be valuable for finding hot code regions. If the tool also includes an execution count number per static region, there is likely some additional data lost that could become useful in the future. The process of creating an abstraction of the dynamic execution by storing only static data and limited contexts limits future analysis to what may be performed using only data the original tool designer found to be relevant.

Constructs Define Hot-Code

The Kremlin tool [31] can find parallel tasks, and generates metrics for the ease of extraction, and the frequency of execution. The ease of extraction metric is mostly a reflection of the number of potential conflicts. The frequency of execution value can help the user focus parallelization efforts only highly executed, or hot, regions of code. Loops created by *for* or *while* constructs are often the source of hot code regions [12]. However, a loop may contain an expensive operation, such as memory allocation or file IO, and may not be executed frequently. Furthermore, a loop may contain instructions that are worthy of parallelization, but are included in a loop that also may conflict with another parallel task. Conflicts make tasks difficult to parallelize; some tools and may reduce the value of a potential parallel task with conflicts, or eliminate such tasks altogether [31].

This thesis presents a system for finding, analyzing, and extracting potential parallel regions from serial codes. The tool presented by this thesis can be used to quickly develop dynamic traces analyses. The trace compression developed for this thesis allows for off-line analysis, thereby allowing new analysis to quickly be tested and executed on a benchmark suite, without the additional step of re-running each benchmark. In fact, the ability to quickly develop new analysis allows this system to be rapidly adapted to new tasks, such as finding irrelevant code in dynamic traces. For modern multi-core systems, the work in this thesis can find parallel tasks outside of artificial constructs at a granularity of hundreds to millions of instructions.

Chapter 3

Related Works

This chapter presents work relevant to this thesis. Trace compression, visualization, and analysis are examined in this chapter. Decision diagram (DD) construction techniques are also discussed.

3.1 Dynamic Trace Compression

Managing large dynamic traces is the key problem when performing whole program analyses. Consider that 1 billion 64bit values need $\approx$ 7.5 GB of uncompressed storage and that traces usually contain billions of instructions (10-1000s of GB). Therefore, any trace analysis tool must operate on compressed data. Further, analyzing this data many need either sequential or random access mandating different compression techniques. ParaMeter requires rapid random access coupled with good compression.

Researchers have used three methods to reduce trace sizes: (1) abstraction; (2) compression by predictor-based encoding; and, (3) compression by program structure exposing methods, such as run-length encoding or hierarchical grammars. Stride-based compression techniques have recently emerged as a viable option for some analyses, such has hot-code and memory-conflict location [50].

3.1.1 Abstraction and Elimination

Data abstraction creates an abstract representation for concrete data. Abstract representations can be more compact and easier to analyze. For example, some abstractions applied to concrete dynamic trace data include, but are not limited to, a mapping from instructions to object creation/destruction [85], or program slices [105].

Data elimination removes information from a trace that is deemed unnecessary by the original analysis designer. For example, the whole program path (WPP) trace representation [58] aggregates data from multiple occurrences of the same program path. However, information that is difficult to aggregate, like cache misses or CPU temperatures, will be removed from the WPP representation.

3.1.2 SEQUITUR Compression

The Sequitur compression algorithm is useful for compressing instruction level dynamic traces into an analyzable representation [67]. However, while Sequitur can rapidly provide an analysis with information about the instruction sequences in a trace, specific instruction information often requires an expensive traversal of the grammar. Do demonstrate grammar creation, lets generate a grammar for the following sequence of letters:

$$abcdabcdabc$$

We begin by creating a start rule:

$$S \rightarrow abcdabcdabc$$

Sequitur forms a grammar online, thus the algorithm would first detect the repetition of the variables ab. The algorithm then creates a new rule $A \rightarrow ab$:

$$S \rightarrow AcdAcdAc$$

$$A \rightarrow ab$$

This processes is repeated for Ac, thereby forming a hierarchical grammar:

$$S \rightarrow BdBdB$$

$$A \rightarrow ab$$

$$B \rightarrow Ac$$

This processes is repeated for Bd:

$$S \rightarrow CCB$$

$$A \rightarrow ab$$

$$B \rightarrow Ac$$

$$C \rightarrow Bd$$

3.1.3 Sequential Compressed Traces

Burtscher *et. al.* [18] in 2005 described a predictor based strategy requiring only mispredictions to be stored in the trace file. The resulting compressed trace file is further compressed with a standard stream compressor such as gzip or bzip2 achieving a 10 times compression factor with rapid streaming decompression. Generating interactive DINxRDY plots from such stream compressed data is impractical as the data must be entirely decompressed for each frame, ≈ 1 minute on a 2.0 GHz Pentium 4 for each 800x600 pixel plot in a trace containing only 100 million instructions. Worse, selecting and performing slice analysis on instructions requires two additional passes through the complete data set adding another 2 minutes to the frame's render time.

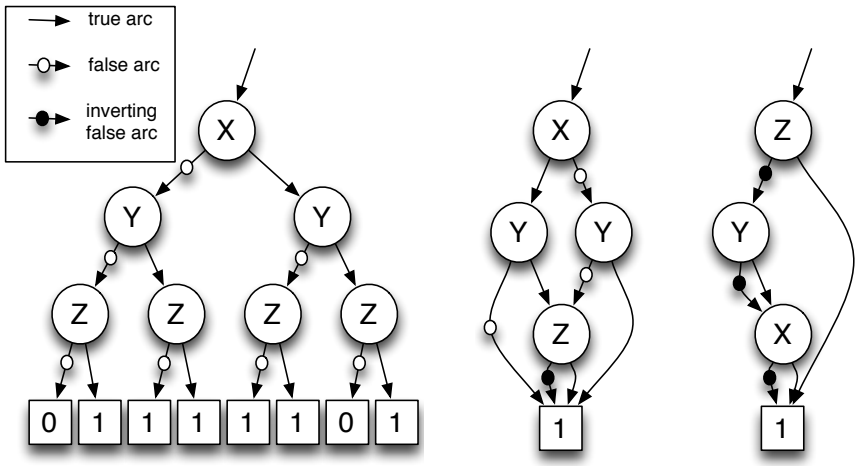
Work by both Iyer *et. al.* [44] and Zhang et al. [105] addresses this problem by generating intermediate representations. Iyer's work maintains a stream compressed intermediate representation suitable for working on the current frame, but leaves the navigation problem unsolved. Zhang et al. [105] use a BDD to maintain the *intermediate analysis* results in a compact form in RAM. However, the navigation problem, as well as inquiries into the contents of a DINxRDY plot, is still prohibitively expensive.

3.2 Reduced Ordered Binary Decision Diagrams

Reduced, ordered, binary decision diagrams (BDD) were first described by Akers [6] and further developed by Bryant [17]. BDDs have been used in many domains including hardware verification [14], cryptography [53], static program analyses [56, 100], and some use in dynamic trace analysis [105].

BDDs can be viewed as compressed versions of binary decision trees. Figure 3.1(a) shows a binary tree for the three variable function $f(x, y, z) = x'y + xy' + z$. For example, traversing the left edges of the graph we evaluate $f(0, 0, 0)$ as 0. BDDs are a graph data structure in which each node corresponds to a Boolean function (just as each node in a binary decision tree) [17]. The following two reduction rules are used to convert a decision tree to an BDd:

(1) When two BDD nodes *p* and *q* are identical, edges leading to *q* are changed to lead to *p* and *q* is removed



(a) Binary Decision Tree (b) Naïve Order (c) Better Order

Figure 3.1: A Three-variable Binary Decision Tree and BDDs

(2) If both edges from a node p go to child node q, then p is eliminated and all nodes that go to p are redirected to q.

The last reduction rule is commonly referred to as the **S-deletion** rule [43]. Figure 3.1(b) shows the BDD for f under the variable ordering (x, y, z) with additional compression provided by inverting edges. To compute $f(0, 0, 0)$ with the BDD, we traverse the 0, or false, arc of the X node, the false arc of the rightmost Y node and the inverting false arc of the Z node. Because we reached the constant 1 node through an odd number of inverting arcs, we find $f(0, 0, 0) = 0$ as before.

3.3 Zero-Suppressed Binary Decision Diagrams

For this research I propose using BDDs to represent sets of dynamic program trace information. The Boolean function that describes the inclusion of a set in a Boolean function is called the *characteristic function*. BDDs can perform many set operations efficiently [17].

However, Minato [42] found that BDDs were inconvenient for sets of binary vectors. The tuple based dynamic trace encoding method used by this proposal employs a variation of this binary vector encoding technique. Specifically, a Minato creates sets of combinations of objects represented by a binary vector, $(x_n x_{n-1} x_{n-2} ... x_2 x_1)$. In this vector each bit represents the inclusion of the object in the set. Using BDDs, each bit in the binary vector and a variable in the Boolean formula. The size of a BDD depends on the number of variables in the encoding, as well as the variable order. Therefore, it is useful to know the smallest number of relevant bits in a bit vector before performing BDD encoding. Unfortunately, it is not always possible to know the smallest number of relevant bits for a bit vector.

The following example better illustrates this problem: The vector 0101 has four bits. If the rightmost bit is the least significant bit, then, in this case, let us assume only the rightmost three bits are actually useful. It is possible to encode this vector in a BDD with three variables, representing the vector 101. However, imagine this operation $101 \wedge 1101$. The BDD must not contain four variables, but Boolean algebra states the vector 101 contains a *don't care* value for the fourth bit, which is not correct. Thus, the BDD should be built with a variable for each potential bit in the bit vector.

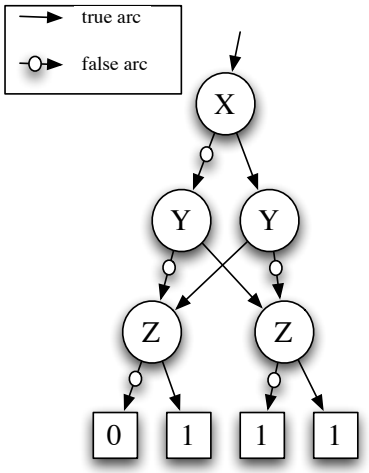


Figure 3.2: A ZDD for $f(x, y, z)$

Minato [42] proposed a variation on BDDs, called ZDDS, to address this issue, and to reduce the size of BDDs for sparse data sets. ZDDs are a variant of BDDs where the *S-deletion* compression rule is replaced by the use of the **pD-deletion** compression rule. In this section we see that ZDDs provide better compression than BDDs for trace data, and over $9\times$ faster creation times.

Zero-suppressed BDDs, or ZDDs, replace the *S-deletion* rule with the *pD-deletion* rule. This rule states the following:

- If the *1* edge from a node *p* leads to a zero terminal node and whose *0* edge a child node *q*, then *p* is eliminated and all nodes that lead to *p* are redirected to *q*.

Furthermore, ZDDs do not typically implement the inverting arcs optimization, i.e., ZDDs have no inverting arcs, only plain $then$ and $else$ arcs. To see how this rule change results in a different decision diagram, consider, once again, the function $f(x, y, z)$ whose binary decision tree was shown in Figure 3.1a. Figure 3.2 shows the ZDD for this function. For this function, ZDDs perform worse than BDDs, as most of the values for (x, y, z) cause the function to evaluate to 1. However, for sparse functions (i.e., those with few 1's in the range), such as trace data, ZDDs provide superior compression.

(a) As Captured (b) Ideal Schedule

Figure 3.3: Basic DINxRDY Plot with 5 instructions.

3.4 Hot Code Analysis

Ammons *et. al.* found that hot code regions often occupy less than 28% of the overall program code, resulting in up to 98% of level one cache misses [10]. Therefore, hot code information can tell programmers where to focus parallelization efforts. Using Sequitur-based trace representations, hot path information can be generated by recursively summing the number of occurrences of sub-rules in a grammar [58].

3.5 Dynamic Trace Analysis

Dynamic traces information consists of information collected from a program at run-time. Software developers can use instruction level dynamic trace analyses to create a picture of their software's run-time behavior, as well as software debugging [103]. However, dynamic traces can grow quickly in size (gigabytes to terabytes) for seconds of program execution. Analysis of large quantities of data can be impractical using commodity hardware [83].

3.6 DINxRDY Visualizations

Though parallelism found in prior studies [11, 55, 96] is not accessible to ILP techniques [96], it may yield to thread level parallel (TLP) techniques. The Dynamic Instruction Number vs. Ready-time (DINxRDY) plot (originally introduced by Postiff et al. [72]) can be useful in identifying the potential TLP inherent in many sequential applications.

DINxRDY plots graphically represent parallel structures and expose potential threads for TLP [44]. Consider the hypothetical 5 instruction trace shown in Figure 3.3(a). The vertical axis represents the Dynamic Instruction Number (DIN) and the horizontal axis represents the earliest time at which an instruction can be scheduled (ready-time, RDY). For example, Figure 3.3(a) shows that the 3rd instruction in the trace (dynamic instruction 3, or DIN 3) was issued in cycle 3. Figure 3.3(b) shows the same trace under an ideal schedule (i.e., one cycle per instruction, perfect branch prediction, infinite hardware resources, etc.) that respects all data dependencies. This plot shows that DIN 3 is dependent on DIN 2 which in turn is dependent on DIN 1. Further, the plot shows that DIN 4 is not dependent on DINs 1, 2, or 3 because it is scheduled in the first cycle. Dependency analysis is needed to decide whether DIN 5 is dependent on DIN 1 or 4.

Iyer *et. al.* observed that lines running from lower left to upper right in DINxRDY plots form dependency chains of relatively nearby instructions in a program [44]. Further, diagonal lines that have overlapping x-extents suggest regions of code that might be convertible to TLP. Figure 3.6 shows a DINxRDY plot for 254.gap with groups of divergent dependency chains (DDCs) (circled in the figure) suitable for TLP extraction analysis. From Iyer's work [44], we know that these suggest the presence of either data parallelism or pipeline parallelism.

The Dynamic Instruction Number vs. Ready-time (DINxRDY) plot (originally introduced by Postiff *et. al.* in 1998 [72]) can be a useful in identifying the potential TLP inherent in many sequential applications. The dynamic instruction number (DIN) is a unique number assigned to each instruction when the instruction executes at run-time. A single static instruction can execute many times during program execution and each occurrence of that static instruction would generate a new DIN. The relation from $SIN \rightarrow DIN$ is also stored.

This thesis uses sets $\{DIN_i, \{DIN_d\}\}$ for program analysis. The $DIN_i \rightarrow \{DIN_d\}$ relation, also referred to as DINxDIN in this thesis, maps an instruction DIN_i to a set of instructions DIN_d. The members of the set DIN_d have a dependency that is resolved by dynamic instruction DIN_i. Thus, it is possible to use the $DIN_i \rightarrow DIN_d$ to identify dependency relationships and perform dependence slicing. A full list of ZDD trace tuple types can be found in Appendix A.

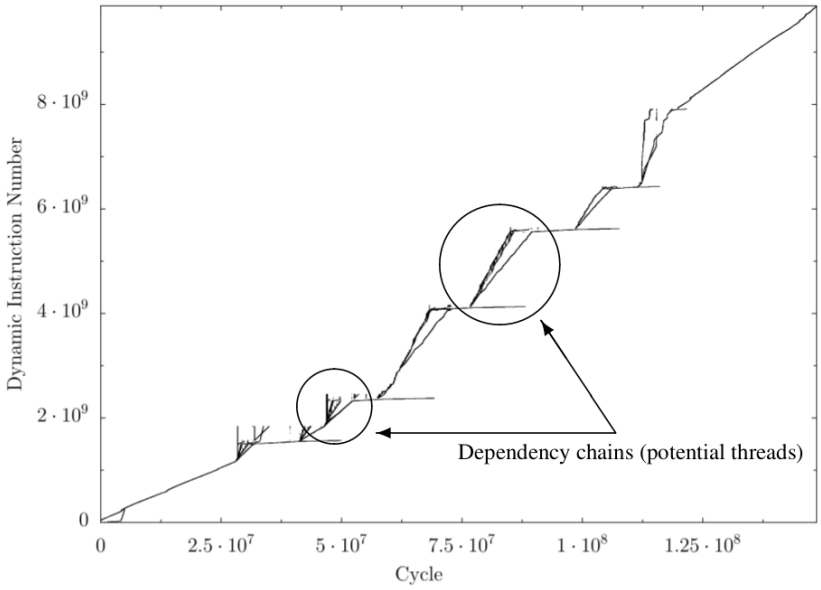


Figure 3.4: Dynamic instruction number vs. ready-time plot of SPEC INT 2000 benchmark 254.gap. Circled areas represent potential threads

3.6.1 Dynamic Program Slicing

Numerous works have used dynamic program trace slicing, introduced by Korel *et. al.* [52], to explore program behavior at run-time. Dynamic slices have been developed for locating the source of observed program bugs [5, 91], program testing [30] and software maintenance [47]. Dynamic traces can quickly grow in size beyond what can be stored and analyzed on commodity hardware [5, 75, 103]. Generating slices from large dynamic traces can require days of computation time [5]. To reduce slice times, this thesis extends the prior use of dependence chopping [36, 54].

Zhang *et. al.* first explored slicing with BDDs [105]. In the work by Zhang, program slice information for each instruction is generated from the dynamic trace information. A variation of technique used to create program slices is also used for the work in this thesis.

3.6.2 Hot Code Analysis

Hot code analysis can tell developers where to focus optimization and parallelization efforts. Ammons *et. al.* found that hot code regions often occupy less than 28% of the overall program code, resulting in up to 98% of level one cache misses [10]. Therefore, hot code information can tell programmers where to focus parallelization efforts. Using Sequitur-based trace representations, hot path information can be generated by recursively summing the number of occurrences of sub-rules in a grammar. Using BDD-encoded traces, hot code information is generated directly from the program execution by maintaining a count of each static instruction's occurrence in the dynamic trace.

3.7 Parallelism in Computing

Until recently, software engineers could gain performance by waiting for processor core performance to improve. However, manufacturers have been unable to extract performance from uni-processor designs, forcing a trend towards multi-core systems [88]. Tasks can be parallelized using a variety of scopes:

- System Parallelism
- Application Parallelism

- Thread Level Parallelism
- Instruction Level Parallelism

With the advent of compute cloud infrastructures, tasks that may be parallelized among systems can be expanded (or reduced) depending on demand [68]. Application parallelism exists if there is a need, or desire, to run multiple applications at the same time on the same operating system. Thread level parallel distributes work between regions of the same program, with no programmer stated total order between regions. Programmers can state an explicit order using synchronization methods such as locks, mutexes or transactions [35, 59].

Limit studies during the 1990s showed that sequential applications have considerable potential parallelism [11, 55, 72, 96] that is amenable to thread level parallelism (TLP) [44] but is not accessible via hardware instruction level parallel techniques (ILP) [96]. Therefore, the proposed research will focus on software optimization through extraction of thread level parallelism from existing sequential applications.

Chapter 4

Dynamic Trace Analysis at Scale

4.1 BDD Compression Time

To understand why BDD-based compression is time consuming, this section begins by first describing BDD-based trace representation [75]. This section then analyzes the BDD-trace creation algorithm and shows that inefficient trace BDD compression is primarily caused by:

- Frequent garbage collection of dead BDD nodes
- The deallocation of potentially reusable dead BDD nodes and corresponding BDD system cache entries

4.1.1 Traces as Boolean Functions

BDDs represent boolean functions, and thus, to represent traces as BDDs, let us review how to represent trace data as a boolean function [75].

Observe that boolean functions can encode arbitrary binary data. For example, if the represented universe, Ω, consists of 4 elements $\Omega = \{a, b, c, d\}$, then a 2-bit encoding can be used to represent each element, $\{a \mapsto 00, b \mapsto 01, c \mapsto 10, d \mapsto 11\}$ [75]. It is possible to create a boolean indicator function (i.e., characteristic function) that evaluates to true for any subset of the Ω with this encoding. For example, the indicator function for the set $\{a, b\}$ is $I_{\{a,b\}} = x'$ where x is the variable for the most-significant bit (MSb) of the set encoding and x' is read as not x.

It is possible to extend this simple notion to encode different data sets necessary for representing and analyzing the trace. All trace data is encoded by a set of tuples $(DIN, data)$ where the DIN is the dynamic instruction number (i.e., the position in the trace, the first instruction has DIN 0, the second DIN 1, and so on). Therefore, a simple instruction trace is encoded as (DIN, PC), where the PC is the program counter value. Similarly, more complex data relationships can be encoded by simply joining the binary representations of arbitrary tuples into a single equation.

This chapter uses three types of data tuples. The first is (DIN, SIN), where the SIN is the static instruction number or PC (Note that this thesis will use PC and SIN interchangeably). The second tuple type is (DIN, DIN), which is used to represent edges of the trace's dynamic data dependence graph. If a decision diagram (DD) encodes an edge set E, if $(10, 24)$ is in the edge set then the 24th instruction in the trace depends on the 10th instruction in the trace. If we wish to know the PC of either of these instructions, we can refer to the (DIN, SIN) tuple set. Finally, this thesis evaluates (DIN, RDY) tuple sets, which encode the ideal schedule for the trace, i.e., for each DIN, RDY is the earliest time a scheduler could execute that DIN given an ideal machine [78]. A description of all tuple members and relation types can be found in Appendix A.

4.1.2 Boolean Functions as BDDs

BDDs can be viewed as compressed versions of binary decision trees. Figure 3.1(a) shows a binary tree for the three variable function $f(x, y, z) = x'y + xy' + z$. For example, traversing the left edges of the graph we evaluate $f(0, 0, 0)$ as 0. BDDs are a graph data structure in which each node corresponds to a boolean function (just as each node in a binary decision tree does) [17]. A full discussion of BDD creation can be found in Chapter 3, but for clarity the two reduction rules for converting a decision tree to an BDD are:

(1) When two BDD nodes *p* and *q* are identical, edges leading to *q* are changed to lead to *p* and *q* is removed

(2) If both edges from a node *p* go to child node *q*, then *p* is eliminated and all nodes that go to *p* are redirected to *q*.

The last reduction rule is commonly referred to as the *S-deletion* rule [43]. Figure 3.1(b) shows the BDD for f under the variable ordering (x, y, z) with additional compression provided by inverting edges. To compute $f(0, 0, 0)$ with the BDD, we traverse the 0, or false, arc of the X node, the false arc of the rightmost Y node and the inverting false arc of the Z node. Because we reached the constant 1 node through an odd number of inverting arcs, we find $f(0, 0, 0) = 0$ as before. BDD creation is covered in more detail in the literature [17, 75, 84].

4.1.3 BDD Unique Tables

Prior work shows how to encode trace data as BDDs. However, the encoding processes can take an unreasonable amount of time. Inefficient encoding is primarily caused by the interaction of garbage collection and BDD system caches, specifically the unique table and the operation cache.

The *S-deletion* rule used to reduce a binary tree into a BDD is realized through the *unique table*. The unique table enforces strong canonicity because each new node has a unique location in the table. If a node is a duplicate of an existing table node (i.e., it represents the same boolean function), the node is reused from the unique table [17]. The unique table also increases the efficiency of BDD creation. If a BDD node already exists the BDD management system, such as CUDD [84] (a state of the art, high-performance, BDD package), saves time by reusing the existing node and avoiding re-computation for the remainder of the nodes below the cached one.

The unique table can be used to tune the overall creation time of the BDD by altering its size. The size of the unique table must at least be large enough to contain all of the live BDD nodes. With CUDD, however, nodes contained in the subtable can also be dead. Upon garbage collection, these dead nodes are added to *death row*, which is an additional cache used to hold recently invalidated nodes. The nodes on death row can also be resurrected and reused.

In addition to a simple node cache, BDD packages, including CUDD, also have an operation cache that caches the results of BDD operations. For example, if one requests a computation of $B \wedge C$ and the result is A, then CUDD will cache that $A = B \wedge C$. If $B \wedge C$ is requested again, it will immediately return BDD A from this cache, and perform the potentially exponential recursion required to recompute A. This

cache gives BDDs their polynomial time complexity [17].

Unfortunately, garbage collection frees the nodes on death row in order to free memory, which, in turn, evicts corresponding results from the operation cache. As we will see, the eviction of useful results caused by accumulation of real garbage ultimately hinders BDD creation efficiency.

4.1.4 Garbage Collection and Compression Time

Garbage collection allows BDD packages to control memory consumption and free the dead nodes on death row. The CUDD package uses a saturating reference counter to keep track of the amount of node use, and to determine if a node is safe to delete. A reference counting system also aids other BDD functions related to automatic variable ordering.

CUDD uses plain pointers to `DdNodes` as a handle to an entire BDDs. If A, B, and C have different pointer values they will represent different BDDs. In the pseudo code presented above, the B and C BDDs have their reference counts increased to prevent them from being garbage collected. B and C are then combined using a boolean $\wedge$ operator to create the new BDD A. A then also has its reference count increased to prevent garbage collection, but the code now decreases the reference count of B and C. If B and C now have reference counts equal to zero they are considered *dead*, and could be removed by garbage collection.

To explain how BDD trace creation produces garbage, we first must consider the BDD creation algorithm. The algorithm is simple. For each tuple, a BDD is created to represent the single element tuple (see the work [75] for details), call it E, and this BDD is OR'ed into the set of all tuples, call it Ω. The pseudo-code for the algorithm using CUDD calls is shown in Figure 4.1.

Notice that at the end of each loop iteration the only live nodes are those that are part of the BDD for the current Ω; all other nodes are marked as dead.

Now, let us examine how this algorithm interacts with the BDD package and its data structures. Let us set $E = \bar{X} \wedge \bar{Y} \wedge \bar{Z}$ which is exactly the shape of a tuple BDD, assuming that we only had 8 possible tuples and thus 3 boolean variables (in practice there are typically 64-128 variables per tuple). In Figure 4.2 we can see the BDD representation of the boolean function for $E = \bar{X} \wedge \bar{Y} \wedge \bar{Z}$.

As shown in the Figure 4.1, the start of the algorithm initializes the trace BDD as empty. After the

```
DdNode * buildSet()
{
  DdNode *Omega = getEmptySetBdd()
  while(!done) {
    DdNode *OmegaOld = Omega;
    DdNode *E = getNextTupleBdd();
    Cudd_Ref(E);
    Omega = Cudd_or(OmegaOld, E);
    Cudd_Ref(Omega);
    Cudd_RecursiveDeref(E);
    Cudd_RecursiveDeref(OmegaOld);
  }
  return Omega;
}
```

Figure 4.1: BDD Build Set

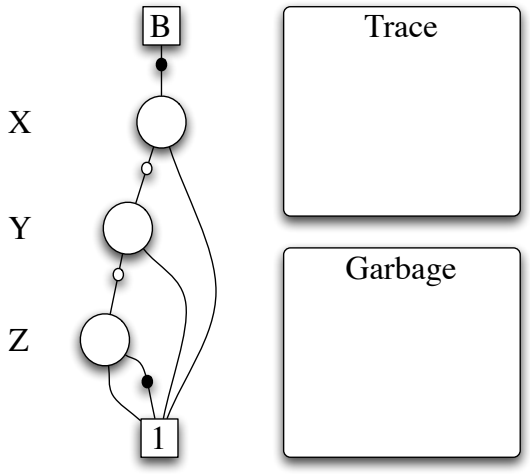


Figure 4.2: A BDD for $\bar{X} \wedge \bar{Y} \wedge \bar{Z}$

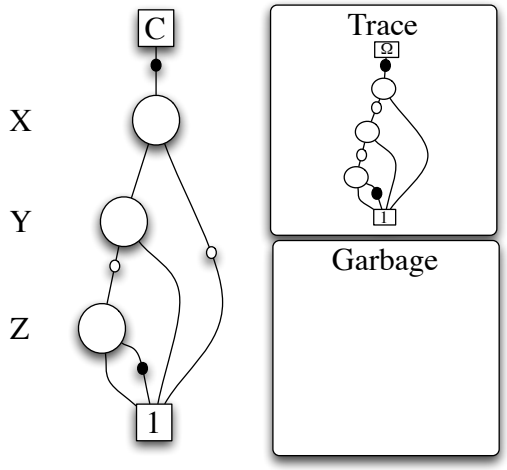


Figure 4.3: A BDD for $(X \wedge \bar{Y} \wedge \bar{Z})$

BDD is created for E, E is added to the trace BDD by computing $\Omega = E \vee 0$, as shown in the pseudo-code in Figure 4.1.

Now we need to add our second tuple to the set, call it E'. The BDD for E' is shown in Figure 4.3 along with the BDD for Ω and the garbage created so far.

Figure 4.4 shows the trace BDD Ω after adding E'. In this new BDD, the X term is now a *don't - care* value because the result of the function no longer depends on the value of X. The *S-deletion* rule removes *don't care* values from the BDD structure. Figure 4.4 shows the BDD for the function $(\bar{X} \wedge \bar{Y} \wedge \bar{Z}) \vee (X \wedge \bar{Y} \wedge \bar{Z})$ with the X node removed.

In Figure 4.5 we show yet another tuple $E'' = \bar{X} \wedge \bar{Y} \wedge \bar{Z}$, which will be added to Ω along with the current trace BDD and the dead BDD nodes. At this point, notice that the entire BDD for both E and E' is dead and will be garbage collected.

Now, notice that E'' is exactly the same as E. However, the BDD for E is garbage and is on death row. If no garbage collection operation has taken place between the creation of E and the creation of E'', the BDD package can quickly resurrect E''. If death row has been cleared by garbage collection, then the BDD for function $\bar{X} \wedge \bar{Y} \wedge \bar{Z}$ must be recreated.

Furthermore, though not shown in this simple example, a similar effect may occur even without repeated tuples. If Ω from prior iterations of the trace creation loop contained sub-BDDs that would be useful

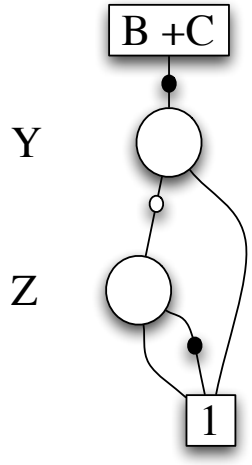


Figure 4.4: A BDD for $(\bar{X} \wedge \bar{Y} \wedge \bar{Z}) \vee (X \wedge \bar{Y} \wedge \bar{Z})$

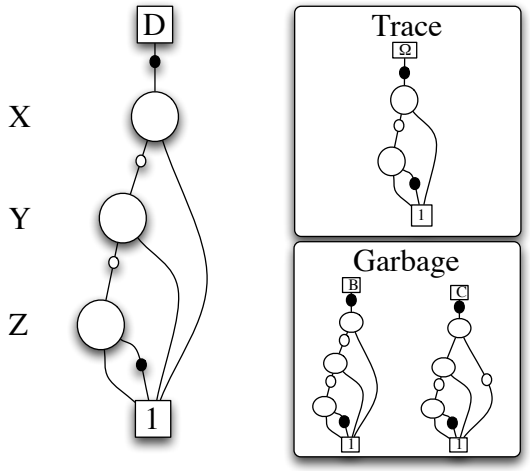


Figure 4.5: A BDD for D

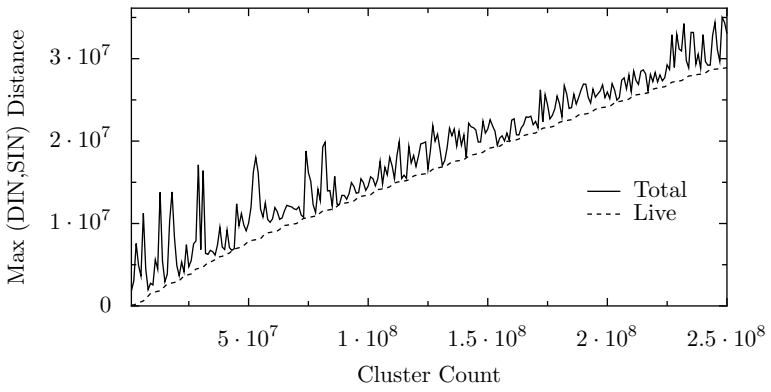


Figure 4.6: BDD Total and Live Nodes

for future $\Omega's$, they too may be garbage collected and thus have to be recreated.

The garbage collection process is triggered when system memory is running low, or when the amount of garbage reaches a threshold set by the BDD package. This threshold is generally tuned to balance garbage collection time and frequency. If the working set of trace-BDD creation is less than the threshold for garbage collection, then the results of many BDD operations will be cached in death row and in the operation cache. However, if the working set size of BDD creation is larger than this threshold, the BDD will free the nodes on death row. Furthermore, if the working set size of the BDD continues to grow throughout trace-BDD creation, then garbage collection will occur more often exacerbating the problem and increasing runtime.

To get an idea of the working set size, we can look at the amount of garbage produced during trace-BDD construction. Figure 4.6 shows the number of live BDD nodes and the total number of BDD nodes present at each sample point in a trace compression run with automatic garbage collection enabled (a run without garbage collection quickly exhausts all system memory) vs. the number of instructions processed in for the (DIN, DIN) BDD for 164.gzip.

Graphs for trace creation in other SPEC INT benchmarks look similar to 164.gzip. From the graph we can see that while the number of live nodes is small, the working set size grows quickly (the spikes in the graph) until automatic garbage collection reclaims the nodes. Because the BDD package must manage this garbage, the package (1) spends most of its time in garbage collection (see Figure 4.7 for a breakdown of garbage collection time vs. total trace creation time), and (2) removes the fraction of nodes that could

accelerate BDD creation during garbage collection.

4.2 ZDD Compressed Traces

ZDDs are a variant of BDDs where the *S-deletion* compression rule is replaced by the use of the *pD-deletion* compression rule. In this section we see that ZDDs provide better compression than BDDs for trace data, and over $9\times$ faster creation times.

4.2.1 BDDs vs. ZDDs

Zero-suppressed BDDs, or ZDDs, replace the *S-deletion* rule with a the *pD-deletion* rule. This rule states the following:

- If the *1* edge from a node *p* leads to a zero terminal node and whose *0* edge a child node *q*, then *p* is eliminated and all nodes that lead to *p* are redirected to *q*.

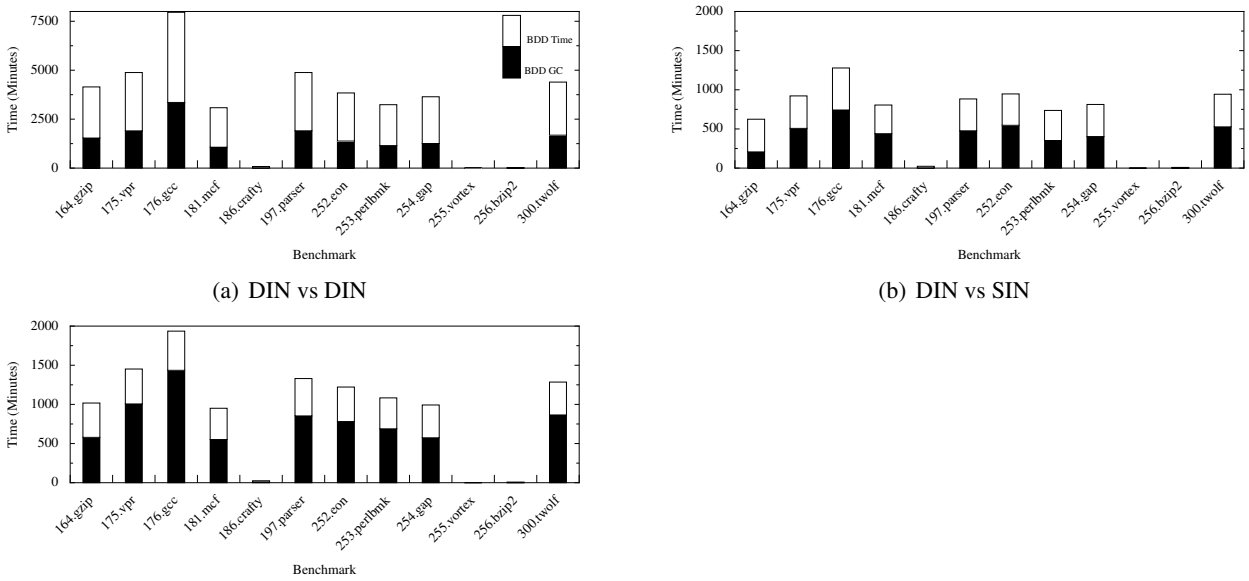
Furthermore, ZDDs do not typically implement the inverting arcs optimization, i.e., ZDDs have no inverting arcs, only plain $then$ and $else$ arcs. To see how this rule change results in a different decision diagram, consider, once again, the function $f(x, y, z)$ whose binary decision tree was shown in Figure 3.1a. Figure 3.2 shows the ZDD for this function.

Note, for this function, ZDDs are bad, as most of the values for (x, y, z) cause the function to evaluate to 1. However, for sparse functions (i.e., those with few 1's in the range), such as trace data, ZDDs are far better.

4.2.2 Traces as ZDDs

To see the advantage of ZDDs for trace creation, let us revisit the equation $E = \bar{X} \wedge \bar{Y} \wedge \bar{Z}$ from Section 4.1. The ZDD, with trace-ZDD and garbage, is shown in Figure 4.8.

ZDD construction does not apply the *S-deletion* rule, therefore the resulting graph does contain Boolean *don't - care* values. However, if a node's $then$ arc terminates at the Boolean false value, that node is removed. In the equation $B = \bar{X} \wedge \bar{Y} \wedge \bar{Z}$ all $then$ arcs terminate at 0, therefore the X, Y and Z nodes are removed,



(a) DIN vs DIN

(b) DIN vs SIN

(c) DIN vs RDY

Figure 4.7: Break-down of Trace Compression time and Garbage Collection time for select SPEC INT benchmarks.

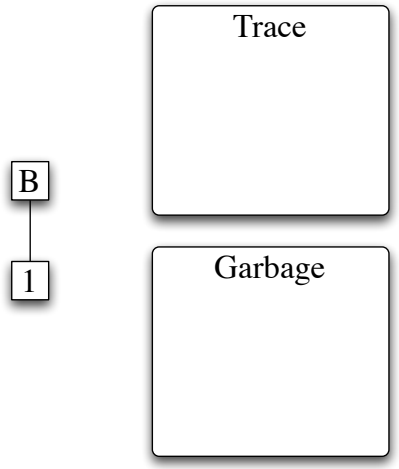


Figure 4.8: A ZDD for $\bar{X} \wedge \bar{Y} \wedge \bar{Z}$

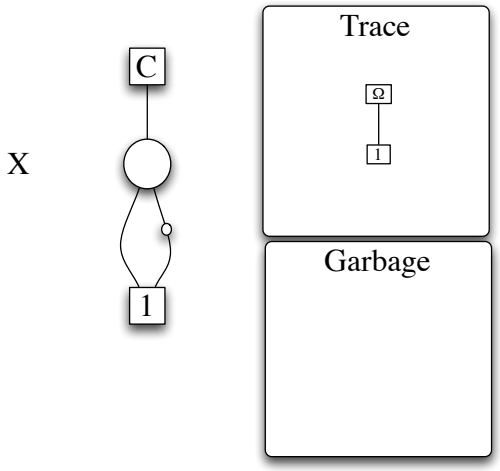


Figure 4.9: A ZDD for $X \wedge \bar{Y} \wedge \bar{Z}$

leaving a very small ZDD. We can now construct the trace ZDD for the equation $E' = X \wedge \bar{Y} \wedge \bar{Z}$, as shown in Figure 4.9.

The ZDD for $E' = X \wedge \bar{Y} \wedge \bar{Z}$ now contains a node for X. Notice X is a boolean *don't-care*, but the BDD *don't-care* reduction rule has been replaced by the ZDD's *zero-suppression* rule. Therefore X is not removed and the $then$ and $else$ arcs point to the same node.

Figure 4.10 shows the state of the trace ZDD after the addition of E'. Notice how little garbage is produced after this addition. Now, like the example from Section 4.1, we can add the function $E'' = \bar{X} \wedge \bar{Y} \wedge \bar{Z}$, where E'' is equal to E. Like trace-BDD creation, the efficiency of this final step will depend if garbage collection has occurred between the creation of E, E', and the trace BDD. However, the ZDD creation of this trace ZDD produced less garbage than the BDD, therefore it is less likely to invoke garbage collection.

In both functions $(\bar{X} \wedge \bar{Y} \wedge \bar{Z})$ and $(X \wedge \bar{Y} \wedge \bar{Z})$ most $then$ arcs terminate at the false, or constant zero node. Note that the trace representation method described in Section 4.1 uses nodes that with zero terminating arcs to represent the binary 0 in a trace. Therefore, as long as the binary representation of such trace data contains many zeros, and is therefore sparse, ZDDs can achieve good compression. In fact, ZDDs have been found achieve better compression levels than BDDs for sets of combinations, as long as the data remains sparse [43].

Figure 4.12, as well as Table 4.1, shows that for same set of traces studied in the work in [78] ZDDs

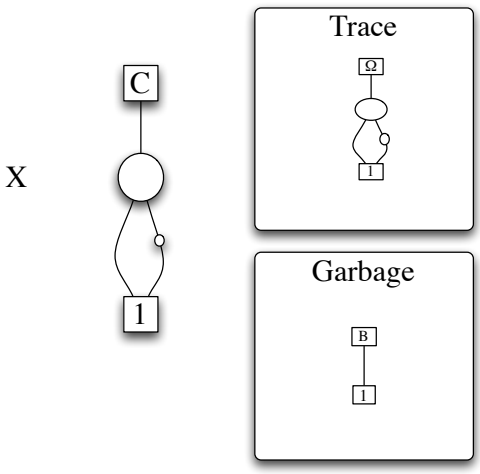


Figure 4.10: A ZDD for $B \wedge C$

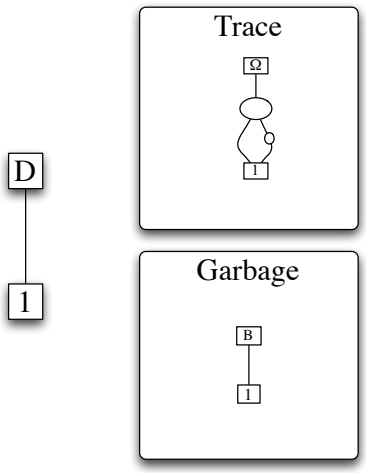


Figure 4.11: A ZDD for $B \wedge C \wedge D$

achieve approximately 25% better compression. Figure 4.12 shows the number of nodes required to represent the BDDs required for various trace analyses and visualizations. The data is for 250 million instruction traces from the SPEC INT 2000 benchmark suite.

4.2.3 ZDD Variable Order, Visualization, and Analysis

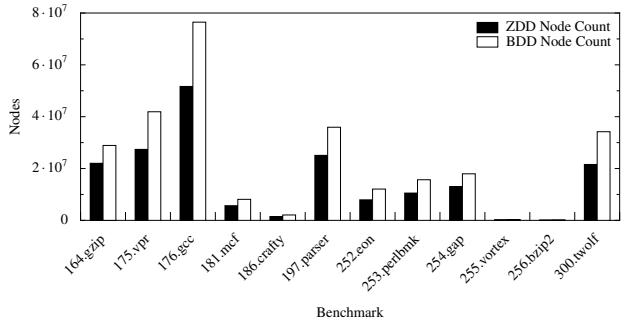
It is important to note that ZDD and BDD size can vary significantly depending on the choice variable order. Furthermore, the choice of the best variable order for a ZDD may not be the same as the best order for the equivalent BDD. This can be problematic, as discussed in Section 4.3, as certain visualization algorithms depend on the variable order in the ZDD. Fortunately, Lhoták *et. al.* found that in many cases the best BDD variable order is also the best ZDD variable order [62]. Lhoták *et. al.* also show that it is trivial to convert any BDD-based program analysis into a ZDD-based analysis. Applying Lhoták *et. al.*'s insight and the concept of BDD-encoded dynamic traces [75], all the standard analyses can also be applied to ZDD compressed traces. The one non-trivial algorithm is the trace visualization algorithm used by the ParaMeter tool [78]. However, Section 4.3 shows how to adapt this algorithm used for ZDDs.

4.2.4 ZDD Compression Time

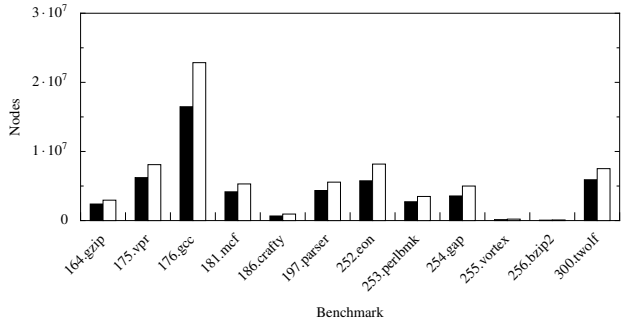
BDD compression time may increase as node count decreases [77]. However, the ZDD unique table growth in Figure 4.13 (compared to Figure 4.6) shows that far less garbage is produced during trace-ZDD creation. Because there is far less garbage produced, the ZDD-based trace compressor spends far less time collecting garbage. Figure 4.14 compares the amount of time required for garbage collection for ZDD and BDD encoded traces for the set of trace data used by ParaMeter [78]. These figures are also presented in Table 4.2. Notice that the ZDD-based code spends far less time collecting garbage.

Now that we know that ZDDs have a small working set size and that the amount of garbage produced stays almost flat during trace-ZDD creation, it is possible to further accelerate ZDD trace creation by adjusting the size of the unique table so that it will contain the working set of the trace-ZDD.

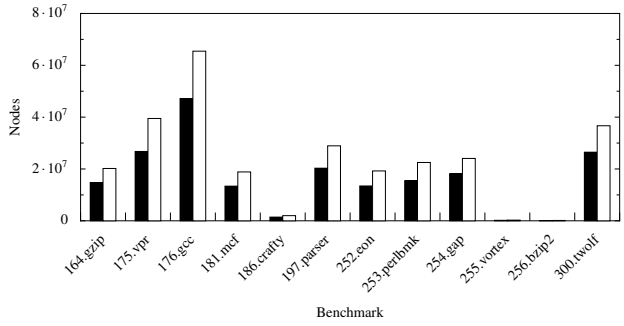
In Figure 4.15, and Table 4.3, we can see the time, in seconds, required to encode 250 million trace instructions into a ZDD compared to the time required by BDDs. In this figure, the size of the unique table



(a) DIN vs DIN Node Count



(b) DIN vs SIN Node Count



(c) DIN vs RDY Node Count

Figure 4.12: DD Node Count

(a) DIN vs RDY

Benchmark	ZDD	BDD
164.gzip	14776031	20191795
175.vpr	26758473	39488720
176.gcc	47191021	65452369
181.mcf	13371337	18857151
186.crafty	1426447	1989019
197.parser	20298664	28928181
252.eon	13438978	19243812
253.perlbmk	15447265	22507110
254.gap	18167754	24067457
255.vortex	193184	260701
256.bzip2	101730	138353
300.twolf	26470707	36674682

(b) DIN vs SIN

Benchmark	ZDD	BDD
164.gzip	2395329	2954172
175.vpr	6220387	8094630
176.gcc	16477762	22851246
181.mcf	4166872	5302426
186.crafty	664726	944032
197.parser	4346929	5561810
252.eon	5758390	8175031
253.perlbmk	2728992	3490785
254.gap	3563179	4998176
255.vortex	140930	212049
256.bzip2	61088	89025
300.twolf	5912991	7515602

(c) DIN vs DIN

Benchmark	ZDD	BDD
164.gzip	22014614	28903561
175.vpr	27382836	41901557
176.gcc	51666471	76472039
181.mcf	5669206	8109996
186.crafty	1438765	2081447
197.parser	25075154	35929906
252.eon	7925540	12075596
253.perlbmk	10525806	15667727
254.gap	13014623	17960509
255.vortex	226433	312175
256.bzip2	113627	157461
300.twolf	21568405	34202383

Full benchmark trace, benchmark ran to completion.

Table 4.1: BDD vs ZDD Node Count

(a) DIN vs RDY

Benchmark	ZDD	BDD
164.gzip	6149	34744
175.vpr	9395	60352
176.gcc	18260	86119
181.mcf	4970	33083
186.crafty	69	505
197.parser	7881	51172
252.eon	6478	46865
253.perlbmk	6417	41291
254.gap	5232	34420
255.vortex	1	10
256.bzip2	14	97
300.twolf	8511	51914

(b) DIN vs SIN

Benchmark	ZDD	BDD
164.gzip	20522	92132
175.vpr	18718	114093
176.gcc	47518	201188
181.mcf	7704	63987
186.crafty	143	1196
197.parser	20720	114329
252.eon	10218	82897
253.perlbmk	9549	68938
254.gap	11944	75353
255.vortex	3	19
256.bzip2	38	216
300.twolf	13663	100911

(c) DIN vs DIN

Benchmark	ZDD	BDD
164.gzip	1964	12410
175.vpr	3879	30428
176.gcc	7184	44513
181.mcf	3447	26368
186.crafty	69	262
197.parser	3042	28573
252.eon	4001	32793
253.perlbmk	2298	21147
254.gap	2547	24172
255.vortex	2	9
256.bzip2	18	90
300.twolf	3315	31596

Full benchmark trace, benchmark ran to completion.

Table 4.2: BDD vs ZDD Garbage Collection Time (Seconds)

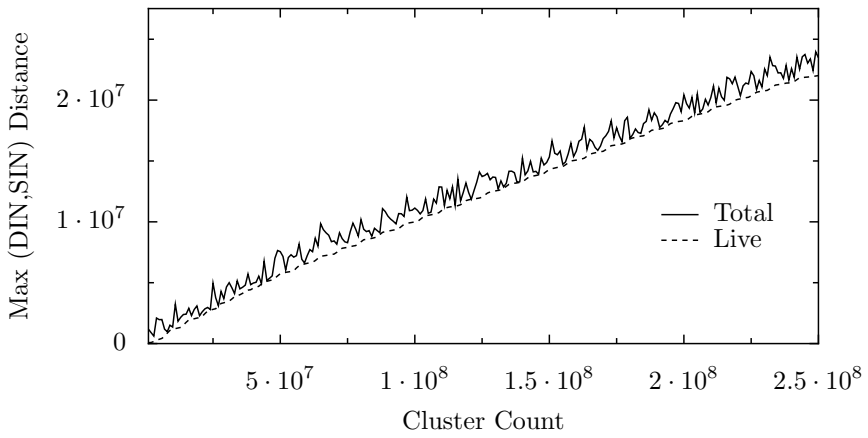


Figure 4.13: ZDD Total and Live Nodes

was manually increased to be initially 100x the normal size for both BDD and ZDD based creation. Note that because BDDs generate so much garbage, their working set size is much larger than available RAM, meaning that they gain little benefit from the 100x increase in table size. ZDDs on the other hand are much faster because the larger unique table can fit almost the entire working set of useful BDD nodes, and garbage does not cause these nodes to be reaped from death row and the operation cache.

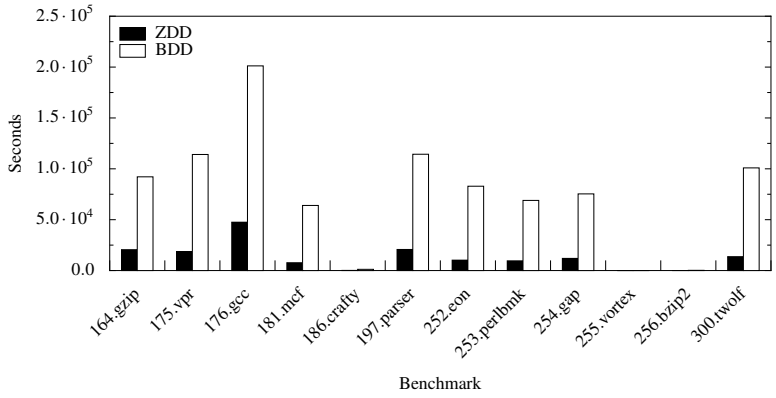
4.3 ZDD Dependence Visualization

Section 4.2 described how to trivially extend BDD-based trace analysis to ZDDs by leveraging prior work. The visualization scheme used by ParaMeter allows interactive identification, analysis, and extraction of parallelism based on BDD-compressed traces [78]. However, this visualization algorithm is specific to BDDs. Given the promise of this approach, we show that it is possible to apply the same with techniques on ZDDs with only slight modifications.

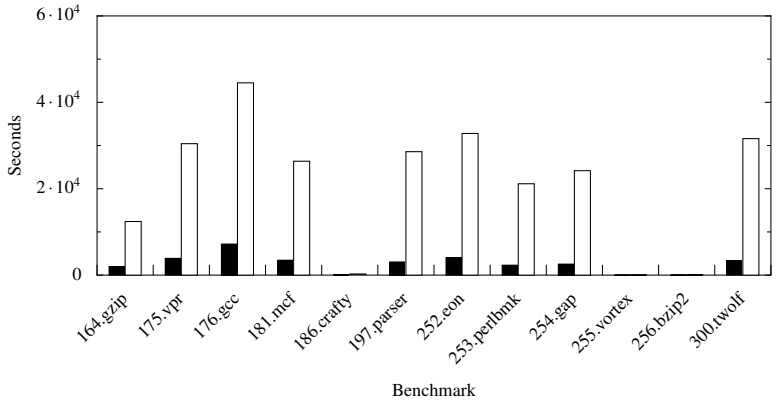
4.3.1 DINxRDY Visualization

The ParaMeter tool, used throughout this thesis, creates a visualization based on the DINxRDY plot, originally introduced by Postiff *et. al.* [72]. An example DINxRDY plot is shown in Figure 3.6.

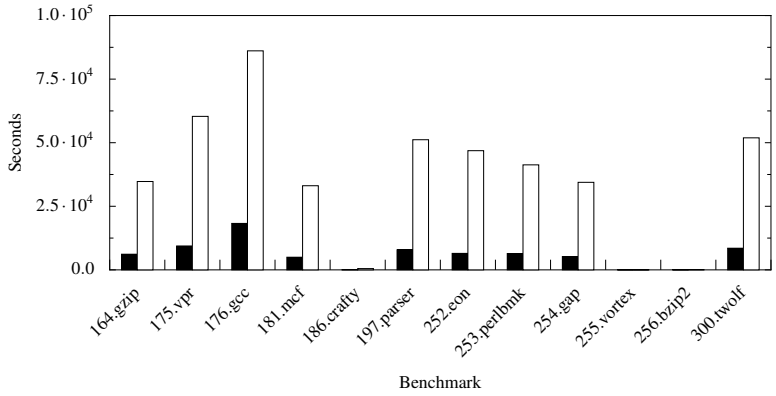
DINxRDY plots can show potential regions of parallelism as follows (as first described by Iyer *et. al.* [44]). Lines in a DINxRDY plot that run from the lower-left to upper right form dependence chains of



(a) DIN vs DIN Garbage Collection Time

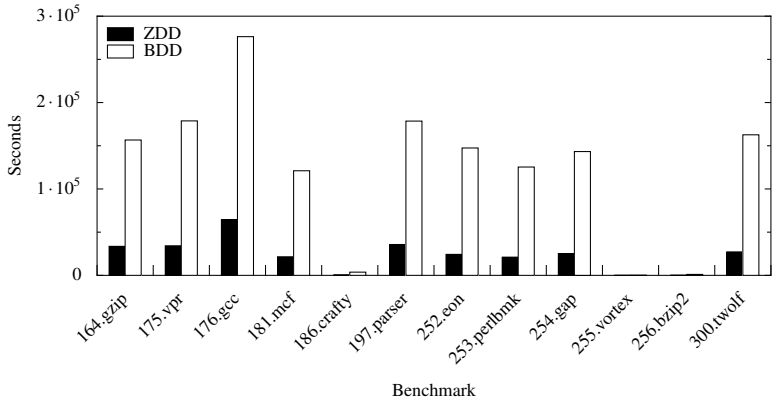


(b) DIN vs SIN Garbage Collection Time

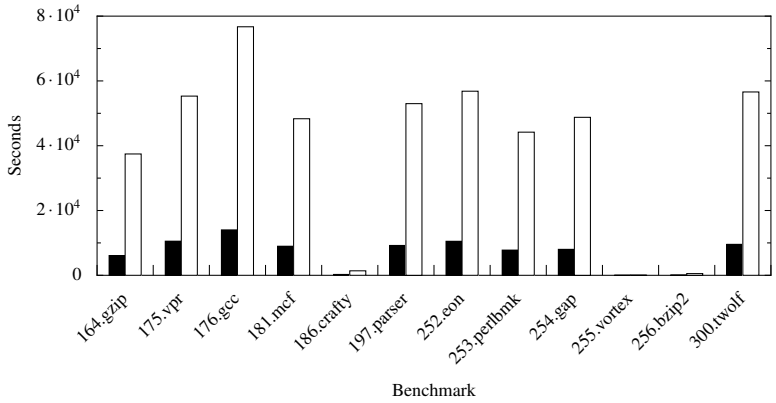


(c) DIN vs RDY Garbage Collection Time

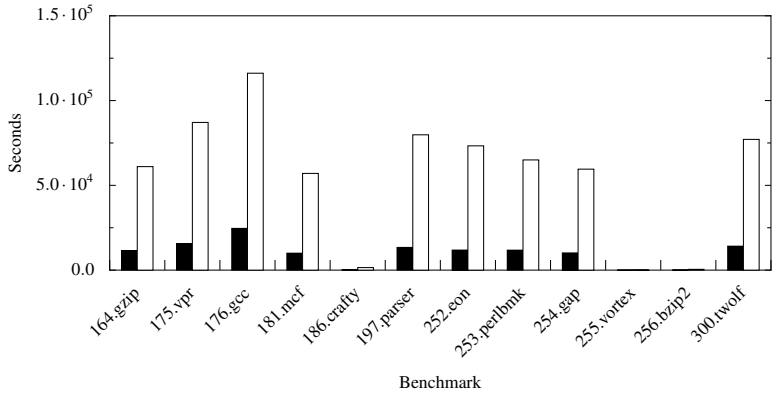
Figure 4.14: DD Garbage Collection Time



(a) DIN vs DIN Creation Time



(b) DIN vs SIN Creation Time



(c) DIN vs RDY Creation Time

Figure 4.15: DD Creation Time

(a) DIN vs RDY

Benchmark	ZDD	BDD
164.gzip	11487	61046
175.vpr	15699	87107
176.gcc	24612	116137
181.mcf	9995	57038
186.crafty	219	1495
197.parser	13392	79809
252.eon	11817	73301
253.perlbmk	11773	64993
254.gap	10145	59557
255.vortex	5	25
256.bzip2	50	469
300.twolf	14125	77106

(b) DIN vs SIN

Benchmark	ZDD	BDD
164.gzip	6100	37454
175.vpr	10532	55314
176.gcc	14015	76713
181.mcf	8974	48328
186.crafty	268	1385
197.parser	9214	52987
252.eon	10503	56826
253.perlbmk	7779	44185
254.gap	8013	48764
255.vortex	7	25
256.bzip2	71	526
300.twolf	9558	56593

(c) DIN vs DIN

Benchmark	ZDD	BDD
164.gzip	33581	156663
175.vpr	34234	178788
176.gcc	64607	276295
181.mcf	21517	121063
186.crafty	521	3711
197.parser	35717	178564
252.eon	24280	147433
253.perlbmk	21072	125391
254.gap	25234	143287
255.vortex	10	61
256.bzip2	132	1099
300.twolf	27167	162679

Full benchmark trace, benchmark ran to completion.

Table 4.3: BDD vs ZDD Creation Time (Seconds)

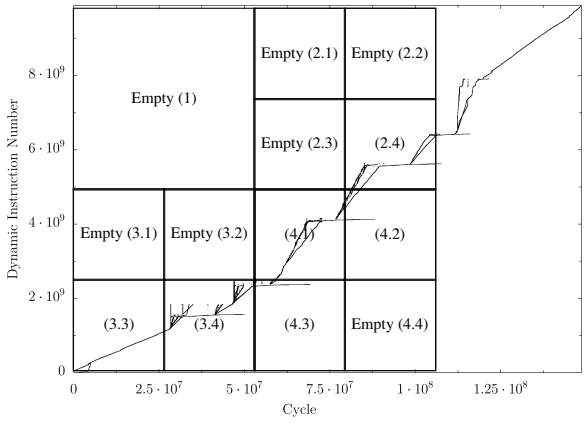


Figure 4.16: Sample partial quad-tree regions superimposed a DINxRDY plot of SPEC INT 2000 benchmark 254.gap. (Not to scale)

relatively nearby instructions [44]. Iyer *et. al.* found that diagonal lines with overlapping x-extents could potentially represent regions of code that have the potential to be parallelized (circled in the Figure 3.6). DINxRDY plots can be used to find and extract parallelism by locating a region in a DINxRDY plot with overlapping x-extents in the 175.vpr benchmark [78]. Using classic program analyses applied to traces, this example extracted both data and pipeline parallelism from the benchmark.

4.3.2 Extended Visualization Algorithm

It is possible to generate visualizations from trace-BDDs in milliseconds by treating the BDD structure like a quad-tree [78]. To understand this algorithm, consider Figures 4.16 and 4.17.

In graphics, a quad-tree decomposes two-dimensional image data into hierarchical regions. Figure 4.17 shows a quad tree for the DINxRDY graph shown in Figure 4.16. The region outlined on Figure 4.16 represents its decomposition, where node Ni corresponds to the region i in the figure.

The visualization algorithm used by ParaMeter is straightforward. Under the variable ordering used in this work, if a BDD is traversed like a tree, then the even levels correspond to bisecting each region horizontally, and the odd levels correspond to bisecting each region vertically. Thus traversing two levels of the BDD corresponds to traversing a single level of the corresponding quad tree. However, since BDDs use the *S-deletion* rule to eliminate nodes, this algorithm must account for eliminated nodes. Since ZDDs use a

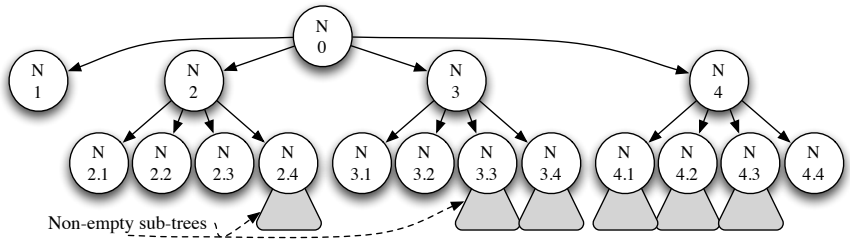


Figure 4.17: Sample partial quad-tree for regions in Figure 4.16

pD-deletion rule, the algorithm for BDDs applies to ZDDs with only a change to how missing levels in the ZDD traversal are handled and what to do when the terminal state is reached.

Recall that the *S-deletion* rule removes nodes from a BDD when both the 1 and 0 outgoing branches lead to the same child node. The quad-tree graphing algorithm must detect removed BDD nodes before graphing the extracted data. For example, consider the indicator function $(\bar{X} \wedge \bar{Y} \wedge \bar{Z}) \vee (X \wedge \bar{Y} \wedge \bar{Z})$. This function represents the set of two numbers $\{000, 100\}$. In the BDD for this function, shown in Figure 4.4 the node for X was removed because it is a boolean *don't care*. Thus, the algorithm used by ParaMeter has to virtually traverse the graph shown in Figure 4.18 instead. To do this, ParaMeter detects that a variable was skipped during traversal and then orchestrates its traversal to virtually traverse outgoing arcs from the removed node, which is shown in grey. If a traversal through the BDD reveals many removed nodes, the number of new arcs grows exponentially in the number of *don't care* values, however, ParaMeter implements a number of optimizations to terminate traversals early, limiting the exponential explosion.

To adapt this algorithm to ZDDs we must understand how to deal with missing nodes. In practice, the final algorithm is simpler than that for BDDs because the *pD-reduction* rule does not have to be undone like the *S-deletion* rule.

In Figure 4.19 we can see the ZDD for the function $(\bar{X} \wedge \bar{Y} \wedge \bar{Z}) \vee (X \wedge \bar{Y} \wedge \bar{Z})$ with removed nodes also highlighted in grey. Notice that, as per the pD-deletion rule, all of the removed nodes have *then* branches leading to the 0 terminal case. Therefore, any time the graphing algorithm detects a removed node, it knows that there is no need to traverse the half of the region where the variable of the missing node is true, and thus needs to do no work. For the half where the variable is false, the algorithm virtually traverses the else-edge of

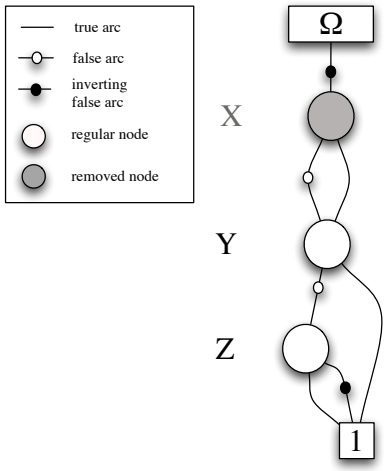


Figure 4.18: Graph BDD With Missing Node

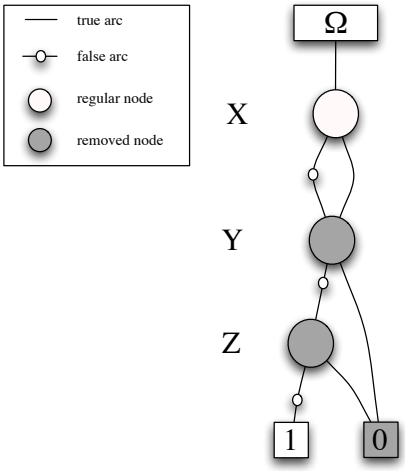


Figure 4.19: Graph ZDD With Missing Nodes

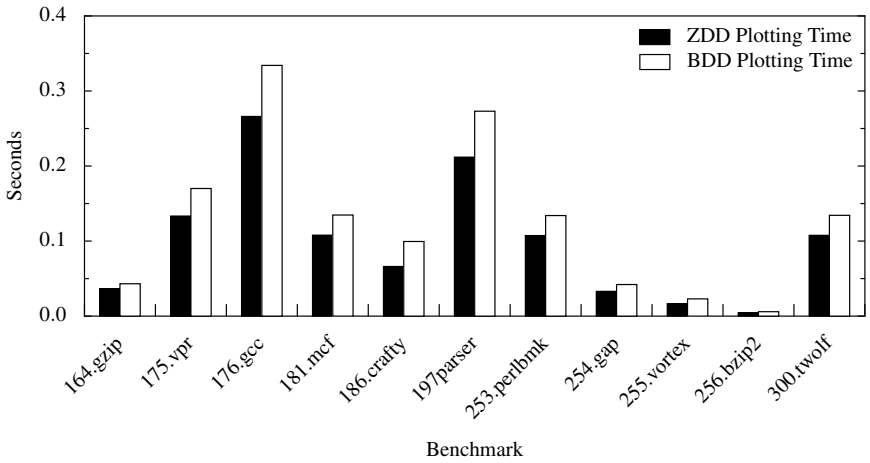


Figure 4.20: ZDD and BDD Visualization Time for SPEC INT 2000 benchmarks

the missing node, just as the BDD-based algorithm traversed both the then and the else edges. Accordingly, Figure 4.20 shows ZDD rendering is slightly faster than BDDs. Data is from a 2.8 GHz Intel Core i7 with 12 GB of RAM running Linux.

Chapter 5

ZDD-based Dynamic Trace Slicing and Chopping

The pseudo code in Figure 5.1 shows how to compute the reverse slice of an instruction with a BDD-encoded DIN d [75]. In this pseudo-code, e is given as the indicator function for the set of edges in the data dependence graph, I_d is given as the indicator function for the instruction with DIN d, and s is the indicator function for the reverse slice that is computed [75]. The variables in indicator function I_d are in the vector $\mathbf{d^1}$, the variables in s are also in vector $\mathbf{d^1}$, and the variables in e are $(\mathbf{d^1}, \mathbf{d^2})$. The function rename takes a function s and renames the variables from set $\mathbf{d^2}$ to the corresponding variables in set $\mathbf{d^1}$.

Slicing depends on the use of existential quantification to remove variables. In the pseudo code in Figure 5.1 the $\exists$ function uses existential quantification to remove the variables e' from the dynamic instructions in $\mathbf{d^2}$. When using BDDs, variables that are removed are interpreted as boolean *don't care* values. However, variables that are removed from a ZDD are interpreted as zero values.

Thus, in order to perform operations that depend on *don't care* values, an additional step is taken to insert *don't care* values. The slicing algorithm shown in Figure 5.3 uses the function $\mathrm{yDC}(d)$ to insert *don't*

function reverse_slice_bdd(e,I_d)
  $s := I_d$
  $s = \mathrm{rename}(\mathbf{d^1}, \mathbf{d^2}, s)$
  $e' := e \wedge s$
  $s := \exists \mathbf{d^2}.e'$
  return s

Figure 5.1: Computing a Reverse Slice using BDDs. [75]

function reverse_slice_zdd(e,I_d)
  $s := I_d$
  $s = \text{rename}(\mathbf{d^1}, \mathbf{d^2}, s)$
  $s = \text{yDC}(s)$
  $e' := e * s$
  $s := \exists \mathbf{d^2}.e'$
  return s

Figure 5.2: Computing a Reverse Slice using ZDD Unate Product.

care values in the y variable positions. This function, given the variable positions for the tuple (x, y) and an identity function d, will return the ZDD dot product [65] of the function d and the universal set of bits for tuple variable y. A similar function, xDC(s), is used for forward slicing.

Let C and D be two trace encoded ZDDs. The unate product of two trace ZDDs is defined as $0\ if\ \exists x(x \in C \cup D\ and\ x' \in C \cup D)$, otherwise $C \cup D$ [37].

Note that the If-Then-Else (ITE) operation can be used to create a logic conjunction. For example, $C \wedge D$ is logically equivalent to $if(C)\ then(D)\ else\ 0$. If x is a literal, for $\exists x \in C \wedge \exists x \in D|$. If $x \notin C \vee x \notin D|0$. Missing variables are treated like 0, and not like a Boolean *don't care*. Thus, the ITE, using bit vectors encoded as ZDDs, captures the logic and behavior used in slicing.

When ZDD functions are unate, the ZDD intersection operation does not take the place of the logic conjunction.

5.1 ZDD Slice Performance

It is possible for a DDG slice to iterate billions of times before reaching a fixed point. Therefore, even a small decrease in slice time can be multiplied into substantial overall increase in performance. This section compares the slicing performance for the *ITE* and the *Intersect* techniques. The *product* slicing algorithm fails to complete in a reasonable amount of time (one week) for these slicing tests. In fact, a *product* based slice fails for DDGs with only 1 million nodes.

function ITE_slice_zdd(e,I_d)
$I'_d = \text{rename}(\mathbf{d^1}, \mathbf{d^2}, I'_d)$
$e' = \text{yDC}(e)$
$s := ite(I'_d, e', zero)$
$s' := \exists \mathbf{d^2}.s$
return s'

Figure 5.3: Computing an Reverse Slice using ZDD ITE.

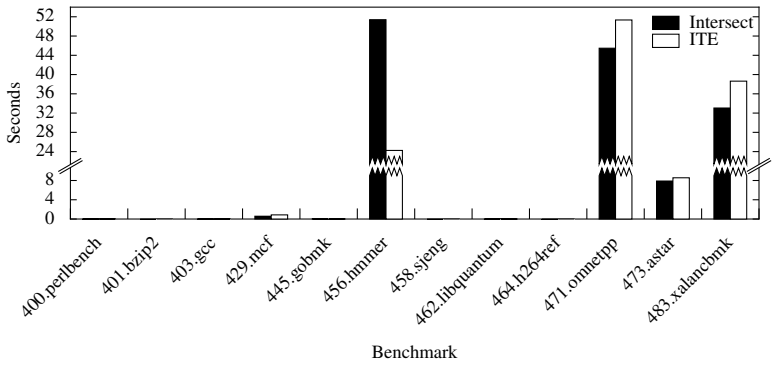
function reverse_slice_zdd(e,I_d)
$s := I_d$
$s = \text{rename}(\mathbf{d^1}, \mathbf{d^2}, s)$
$s = \text{yDC}(s)$
$e' := e \cap s$
$s := \exists \mathbf{d^2}.e'$
return s

Figure 5.4: Computing an Reverse Slice using ZDD Intersection.

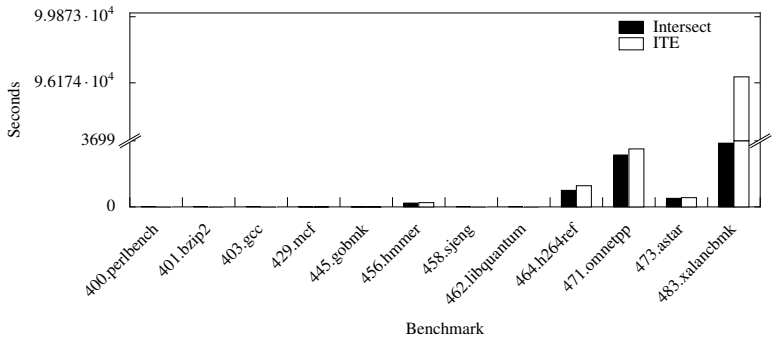
5.1.1 Experimental Setup

Each ZDD slicing technique was tested using a 2.6 GHz Xeon server with 32 GB of RAM. Each slice starts from an initial set of 10, 100, 1000, or 10000 randomly selected instructions from a set of 1 billion instructions. Each slicing operation was repeated six to ten times; the final results in Figure 5.1.1 shows the arithmetic mean.

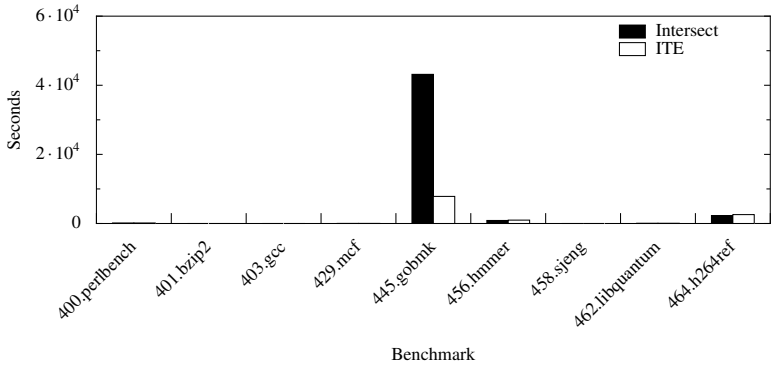
These results show that for most operations ZDD slicing via intersection performs better than the ITE method. Therefore, other analyses in this thesis that depend on a program slice will use intersection. The intersection technique still requires more steps than the comparable BDD slice. Analyses in this thesis that use slicing would likely benefit from a unification of operations used for ZDD slicing, similar to the work by Lhotak *et. al.* with relational operations used for points-to analysis [62].



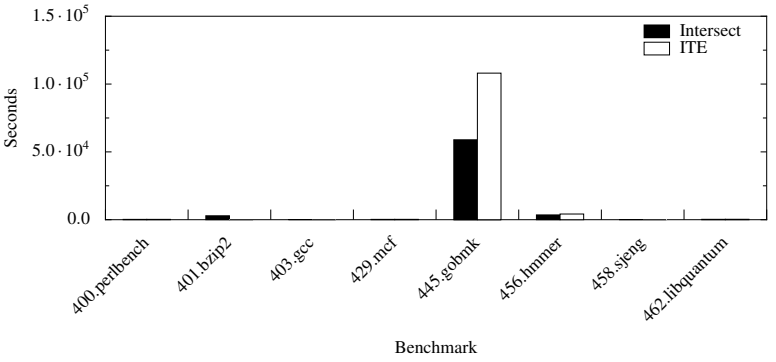
(a) 10 Initial Instructions



(b) 100 Initial Instructions



(c) 1000 Initial Instructions



(d) 10000 Initial Instructions

Figure 5.5: Slicing Times

Chapter 6

Irrelevant Component Elimination

Irrelevant component elimination was used in prior works for removing components from high-level program abstractions [23]. For this thesis, a component is the dependency relation $DIN_i \rightarrow DIN_d$, which maps an instruction DIN_i to instructions, DIN_d, that produce a value DIN_i required for correct execution. The $DIN_i \rightarrow DIN_d$ relation, which also forms a dynamic dependence graph, is stored in sets of $\{DIN_i, DIN_d\}$.

Irrelevant instruction elimination removes instructions from a set d_1 whose forward slice does not reach any instruction in the set d_2. Defining the members of the set d_2 will alter the outcome of an analysis. For example, din-ready analysis creates the set d_2 by computing the set of instructions in the final RDY position in the set $\{DIN, RDY\}$.

The whole-trace irrelevant instruction elimination adds an additional set to d_2. This set contains any dynamic instruction that produces an output value by invoking a Linux system call. An instruction is considered irrelevant if:

- An instructions from a set d_1 whose forward slice does not reach any instruction in the set of instructions in the final RDY position in the set $\{DIN, RDY\}$.
- An instructions from a set d_1 whose forward slice does not reach any instruction in the set of instructions that produce an output through a Linux system call, such as *fwrite*.

It is possible for instructions removed by irrelevant instruction elimination to be the source of useful output. However, irrelevant instruction captures the notion of program execution for cases when a program

function dead_ready_slice(e_{ddg},e_{dinrdy},I_d)
$e_d := \text{yDC}(I_d) \cup e$
$e_f := \text{iter\_slice\_zdd}(e, e_d, \text{forward\_edge\_slice}, 0)$
$topReady = \text{get\_tuple\_top\_y}(e_{dinrdy})$
$I_{topready} = \text{build\_tuple}(topReady)$
$s_{rev} := \text{iter\_slice\_zdd}(I_{topready}, e_f, \text{reverse\_slice\_zdd}, 0)$
$s_{dead} := I_d \setminus s_{rev}$
$e_{notDead} := \Omega \setminus s_{dead}$
$e := e_{dinrdy} \cup e_{notdead}$
return e

Figure 6.1: Irrelevant Instruction Dependency Elimination Pseudo Code

first performs a series of tasks while producing output to the user using system calls, or outputs values to the user at the end of the program execution using some other means. A future extension to dead-ready analysis could allowing the user to define where valuable output takes place in the source code by using the $\{DIN, SIN\}$ edge set. The static instruction number contained in the $\{DIN, SIN\}$ set can be used to lookup source code [78].

Figure 6.1 shows pseudo-code for irrelevant instruction dependency slice algorithm. This algorithm contains new functions $get\_tuple\_top\_y$, $build\_tuple$, and the set Ω. The function $get\_tuple\_top\_y$ returns the value of the largest value of the y variable position, as an integer. The function $build\_tuple$ builds a ZDD for the identity function as $I_{topReady}$. Ω contains a ZDD that represents the identity function for all possible bit combinations in our universe. Thus, in order to create the ZDD universal set Ω, the dead-ready computation must know the maximum number of variables that could exist. ParaMeter defines this number to be 128 in order to represent two 64 bit tuple members [76].

Additional results presented in this chapter compare irrelevant instruction dependencies for programs compiled using the *-O0* and *-O2* optimization settings in the *gcc 4.2.4* compiler. Comparing an optimized binary to a binary without static optimization presents a clearer view of the number of instructions that are irrelevant. Furthermore, this technique can be used to evaluate the effectiveness of a compiler optimization.

Figure 6.17 compares the absolute number of removed dependencies from *-O0* and *-O2*.

Figure 6.16 presents the difference in the irrelevant dependence count from *-O0* to *-O2*. Note that

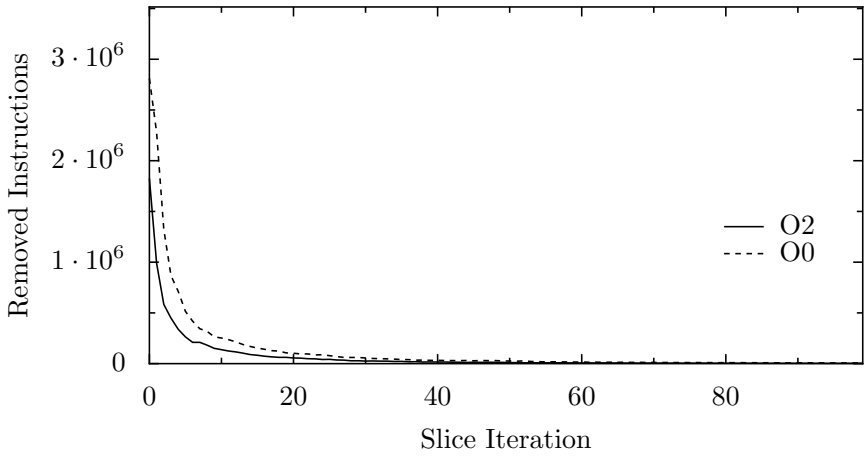


Figure 6.2: 400.perlbench Instruction Dependencies Removed per Iteration

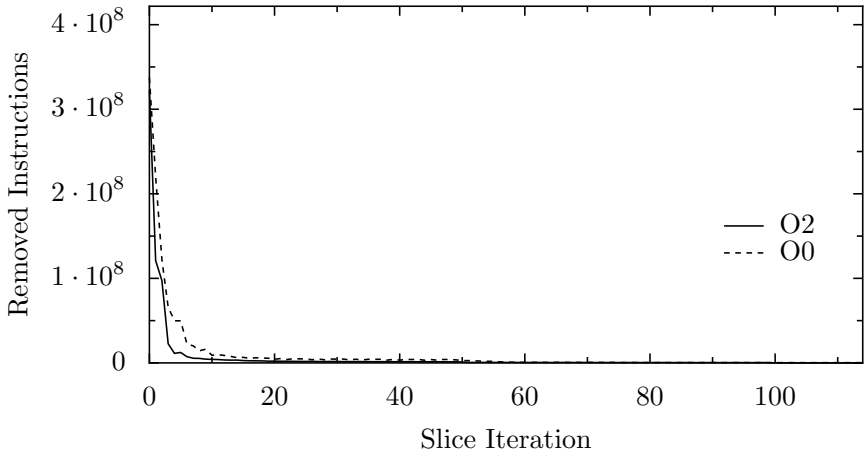


Figure 6.3: 401.bzip2 Instruction Dependencies Removed per Iteration

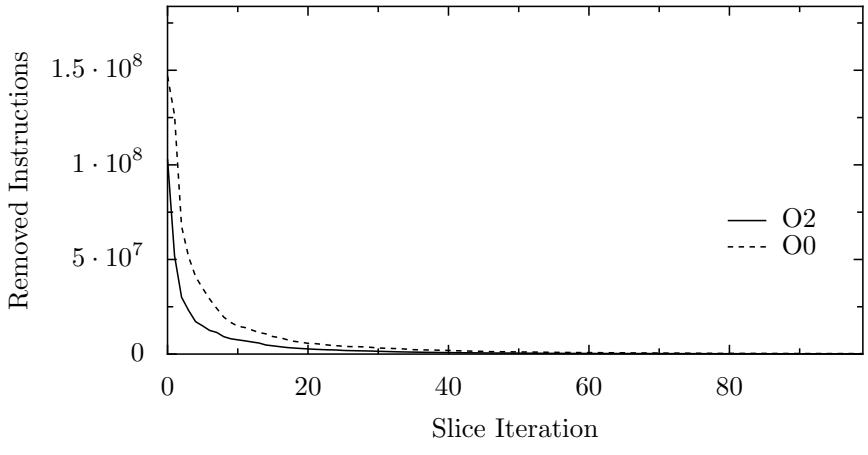


Figure 6.4: 403.gcc Instruction Dependencies Removed per Iteration

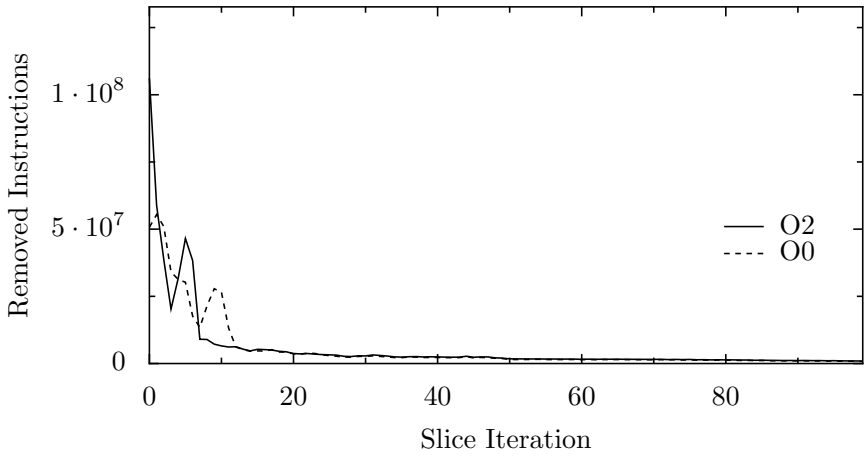


Figure 6.5: 429.mcf Instruction Dependencies Removed per Iteration

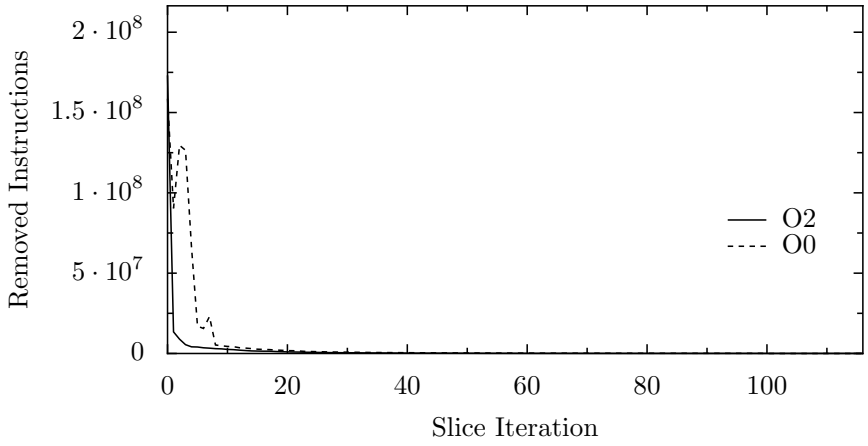


Figure 6.6: 445.gobmk Instruction Dependencies Removed per Iteration

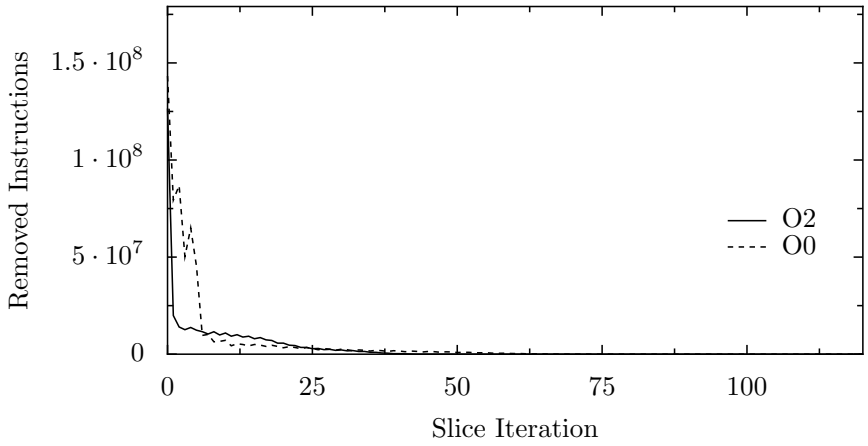


Figure 6.7: 456.hmmer Instruction Dependencies Removed per Iteration

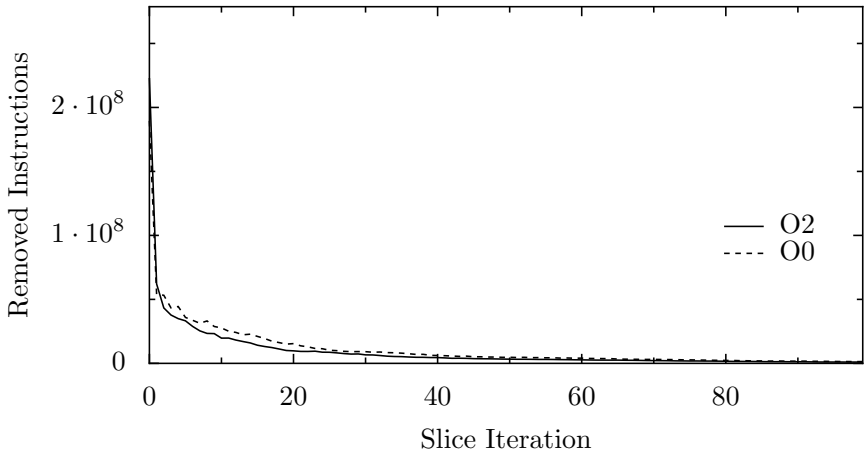


Figure 6.8: 458.sjeng Instruction Dependencies Removed per Iteration

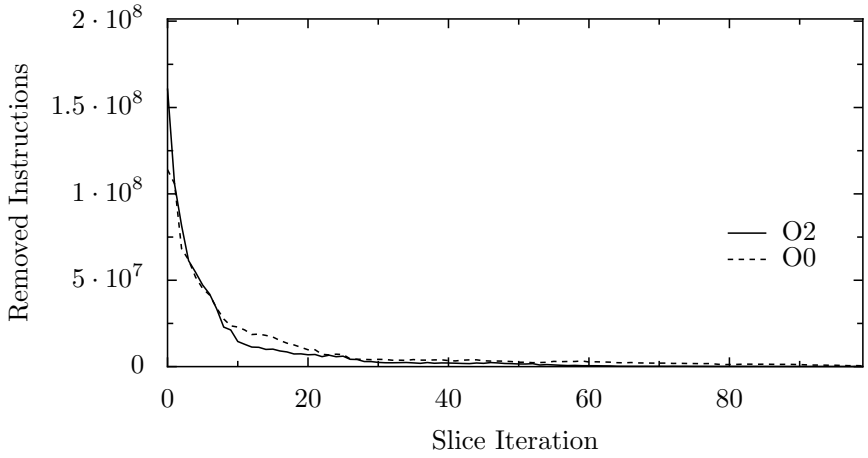


Figure 6.9: 471.omnetpp Instruction Dependencies Removed per Iteration

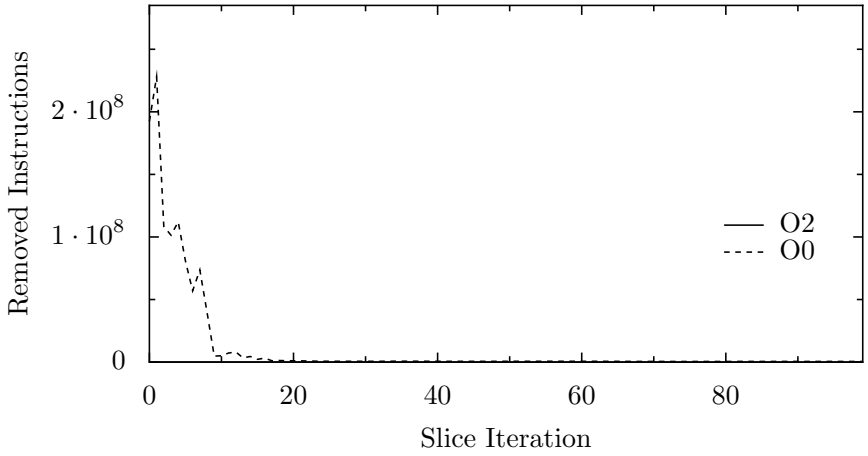


Figure 6.10: 473.astar Instruction Dependencies Removed per Iteration

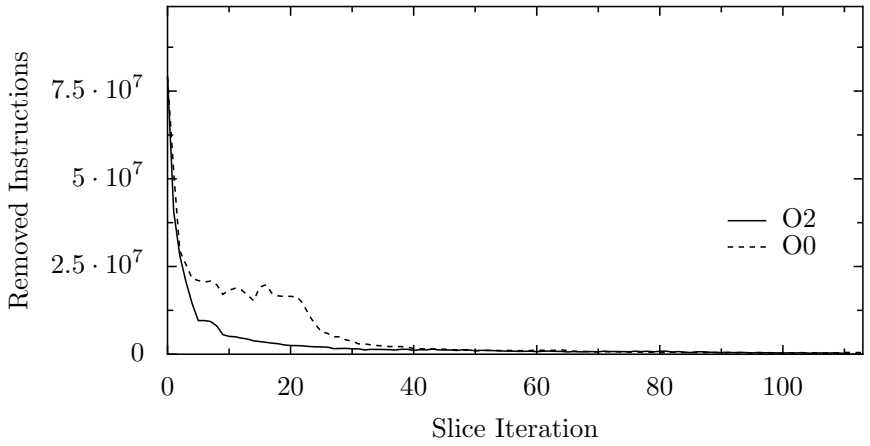


Figure 6.11: 483.xalancbmk Instruction Dependencies Removed per Iteration

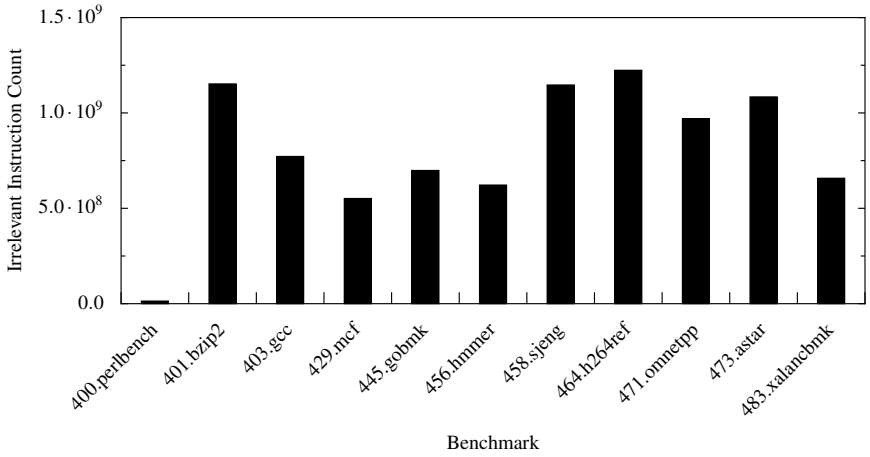


Figure 6.12: Total Irrelevant Instruction Dependencies Count for GCC -O0

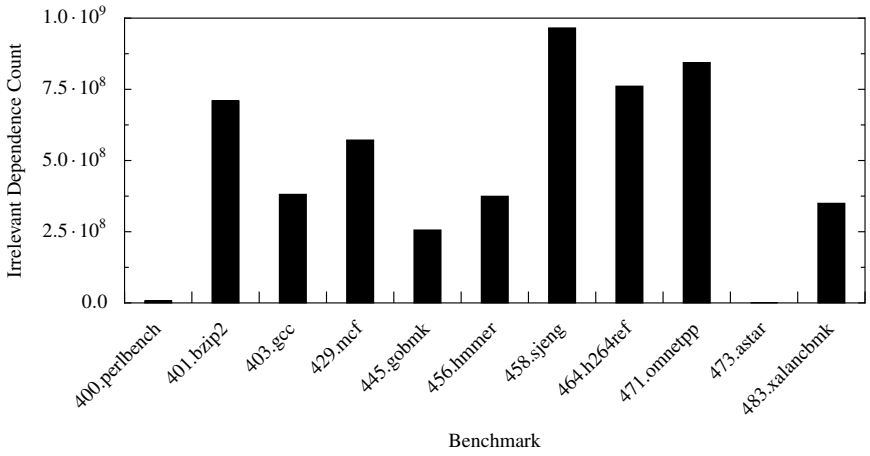


Figure 6.13: Total Irrelevant Instruction Dependencies Count for GCC -O2

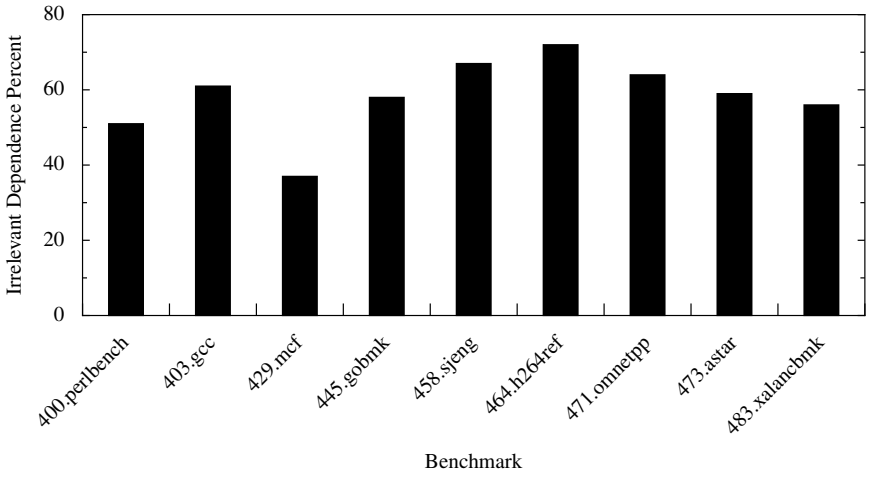


Figure 6.14: Total Irrelevant Instruction Dependencies Percentage for GCC -O0

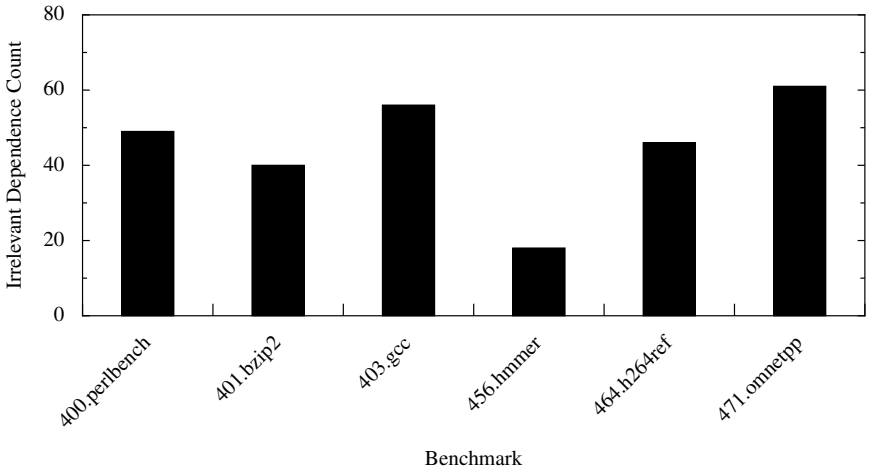


Figure 6.15: Total Irrelevant Instruction Dependence Percentage for GCC -O2

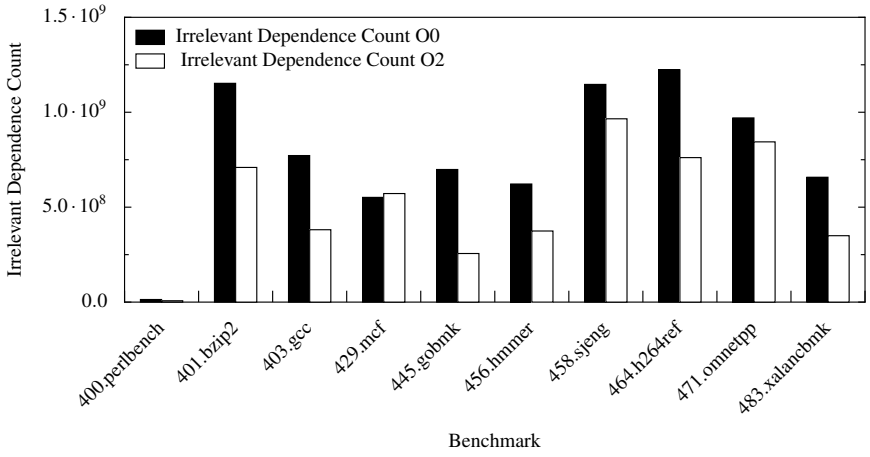


Figure 6.16: Total Irrelevant Instruction Dependence Diff -O0 to -O2

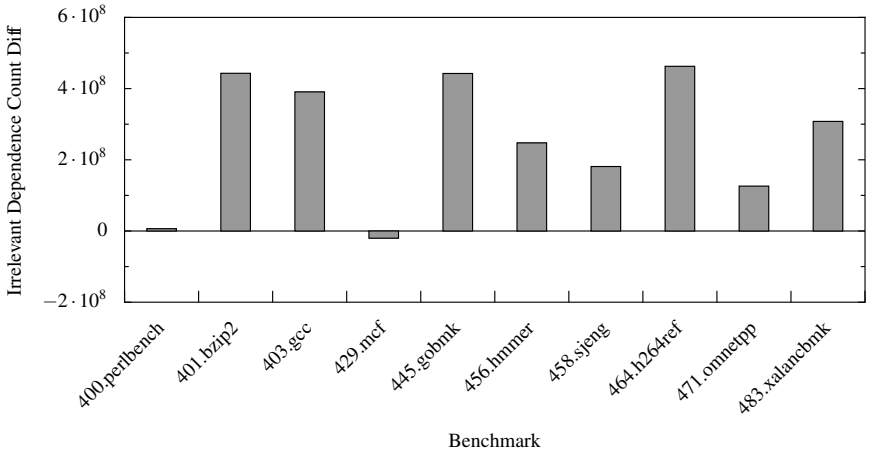


Figure 6.17: Total Irrelevant Instruction Dependence Diff -O0 to -O2

static compiler techniques are able to remove an average of 258,791,915 more instruction dependencies in the *-O0* binary trace than the *-O2* trace.

6.1 Visualization Filtering

Irrelevant instruction dependence elimination uses a ZDD-based DDG chop to find instructions that have no obvious impact on program output. The DDG chop can use any two arbitrary sets of instructions.

DINxRDY time plots can help identify potential parallel tasks [78]. However, some programs produce DINxRDY plots are visually crowded, and thus it can be difficult to identify divergent dynamic decencies chains that are candidates for further parallelization efforts.

This section chops from *hot-code* to *final ready time* instructions, *all instructions* to *final ready time*, as well as the irrelevant instruction chop, implemented as visualization filters in the ParaMeter DINxRDY plot. Furthermore, this section presents a case study that applies each filter to the of the DINxRDY plot of 254.gap from the SPEC 2000 integer benchmark.

6.1.1 Experimental Setup

The visualizations and analyses performed in the study presented in this section were performed using a 1.0 to 1.2 GHz Opteron machine with 17 to 36 GB of memory using the Amazon EC2 [9]. The traces used

in this case study 1 billion dynamic instructions of the 254.gap benchmark using the first reference input. Figure 6.18 shows the DINxRDY plot produced by ParaMeter for 254.gap.

Note that this figures shows two regions in the DINxRDY plot that may correspond to program phases. The first region, labeled α contains a series of instruction chains extending from the bottom-left corner of the DINxRDY plot. Note that these chains are regions of potential parallel execution [78]. The second section, β contains a sharp increase in the slope of the DINxRDY line towards the right side of the plot. The increase in slope is a result of in increase in the number of dynamic instructions that may execute at that same ready time.

6.1.2 Ready Filter

It is possible to only include the set of instructions located at the final ready time for the d_2 set used by the chopping algorithm. This method, called the Dead-Ready filter, performs irrelevant instruction elimination is used to filter a part of the α region of 254.gap. The selection to be filtered is shown in Figure 6.19.

The filtered visualization is shown in Figure 6.20. Note that very few instructions have a forward slice that extends to the end of the DDG. ParaMeter also contains functions necessary to examine source code from a DINxRDY plot [78]. The source code responsible for the region selected for this test includes lines that produced Sum, Diff, and Product functions in the source code file eval.c. The results from these source code lines is then printed and never read again, thus ending the forward chain of dependence.

The second test of dead-ready elimination involves the region in Figure 6.18 labeled β. The selected region of β can be found in Figure 6.21

The results of the dead-ready filter for selected region in β show very few instructions removed, as can be seen in Figure 6.22. Thus, almost all instructions contain a forward slice that reaches to the last RDY time. An inspection of the source code responsible for the instructions in β found that almost all instructions came from the 254.gap memory management system in gasman.c, including the functions InitGasMan and NewBag. These instructions generate memory locations that are used for future operations, and contain a long forward slice until the last RDY time.

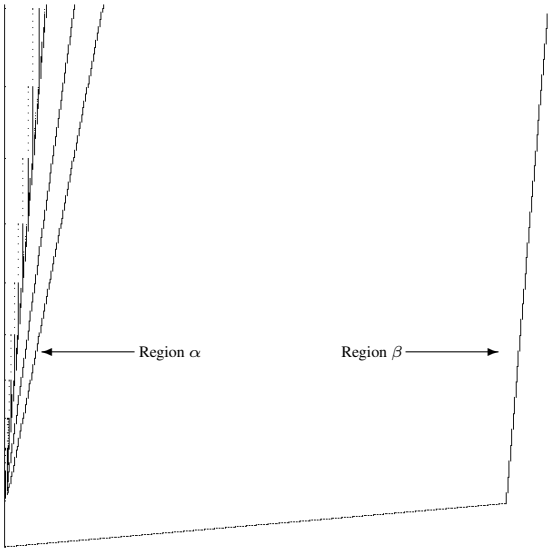


Figure 6.18: DINxRDY plot of SPEC CINT 2000 benchmark 254.gap

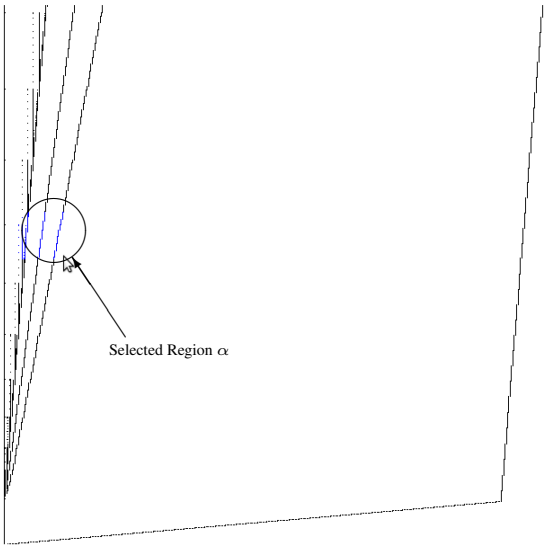


Figure 6.19: Selected Instructions for Dead-Ready Test α

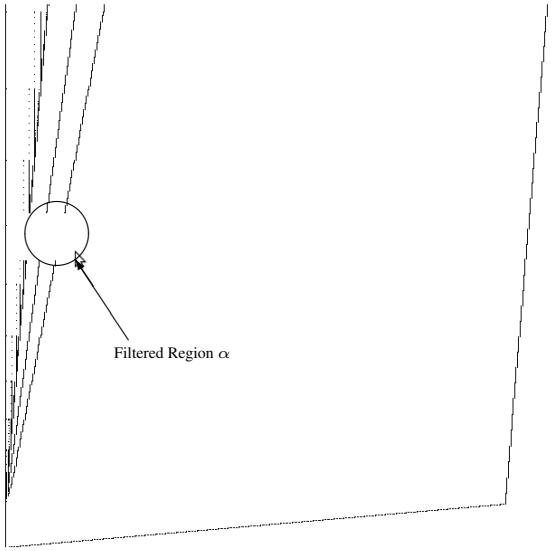


Figure 6.20: Region after Dead-Ready Test of α

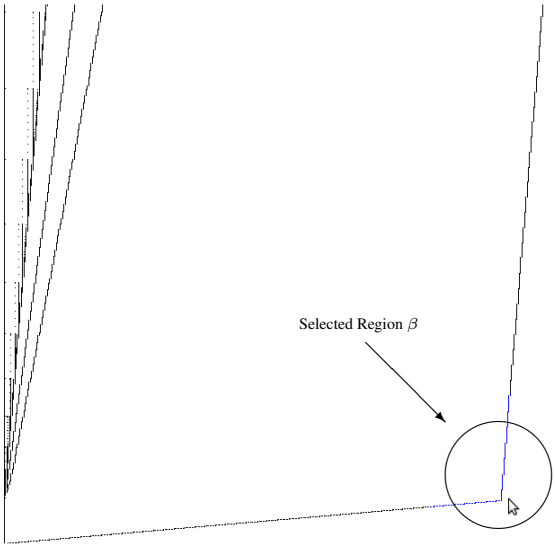


Figure 6.21: Selected Instructions for Dead-Ready Test β

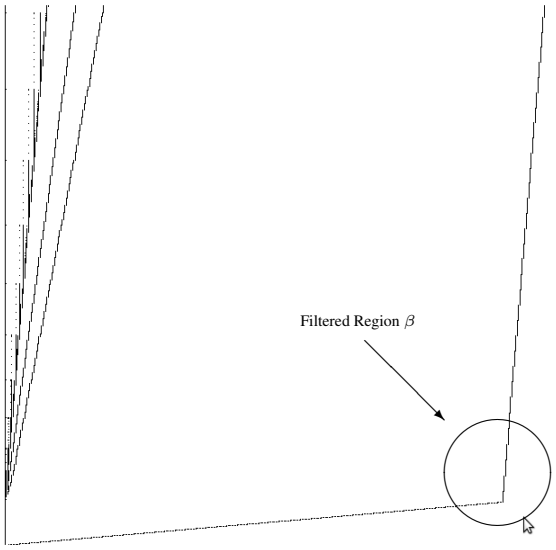


Figure 6.22: Filtered for Dead-Ready Test β

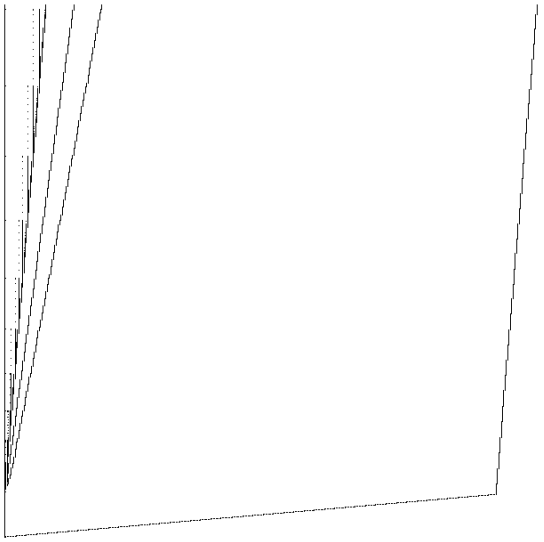


Figure 6.23: 254.gap Filtered with Irrelevant Component Elimination

6.1.3 Irrelevant Instruction Filter

Irrelevant instruction dependence elimination creates a new edge set e_f using a forward slice of the DDG e. Then perform a reverse slice from the set of dynamic instructions collected at the last iteration of the forward slice, d_1. The reverse slice is performed through the set e. It is possible to find the set of instructions whose influence dies immediately by reducing the number of forward slice iterations to one.

The (DIN, RDY) result set created by irrelevant instruction analysis is then removed from the (DIN, RDY) set used by ParaMeter for visualization. This filter was applied to the entire plot of 1 billion instructions from 254.gap. The resulting plot, shown in Figure 6.23, has 72140005 (DIN, RDY) fewer edges than the original plot. However, the visualization shown in Figure 6.23 and the original plot (Figure 6.18) appear almost identical.

6.1.4 Dead-Hot Filter

Figure 6.24 contains the DINxRDY plot for 254.gap with hot code highlighted in red. The Dead-Hot filtering method considers only the top percentage of hot code as the initial starting point for dead-ready analysis. For the case study, the hotness threshold was initially set to 25%, then reduced by increments of one

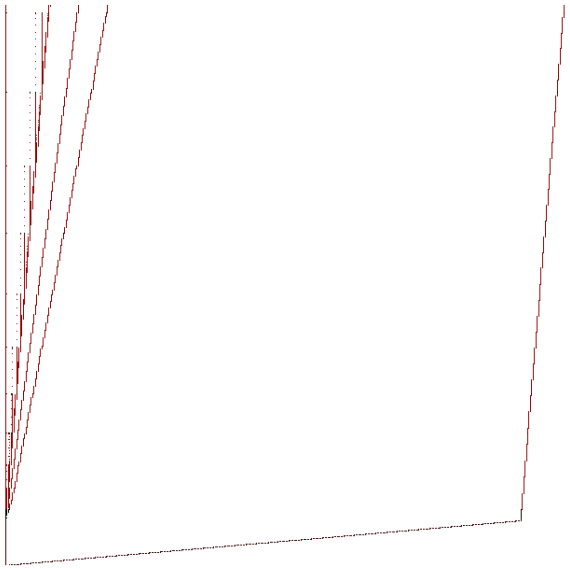


Figure 6.24: 254.gap with Highlighted Hot Code

until the result from dead-ready elimination was greater than the empty set.

Using dead-hot elimination as a visual filter for the entire plot of 254.gap revealed 1) The top 57% of hot codes are eliminated through dead-ready analysis, and 2) the top 58% of hot codes requires days of dead-ready analysis computation to converge. Thus, this implementation of dead-hot allows the user to select a smaller region of code to begin dead-hot analysis. The top 25% of the hottest codes in that region were than used in dead-hot elimination. Dead-ready analysis was unable to remove many instructions from region β, thus we used that region for this section of the case study. The selected region can be seen in Figure 6.25.

The selected region of β contained 2123492 dynamic instructions. The top 25% of the hot code from this region contains 1068749 dynamic instructions. However, Figure 6.26 and Figure 6.26 show that the bottom 75% of the hot code is responsible for almost all of the DINxRDY visualization of region β.

The dead-ready analysis produced 200376 dynamic instructions for this example. The total time for the analysis was approximately 20 Minutes. The final filtered image is shown in Figure 6.27.

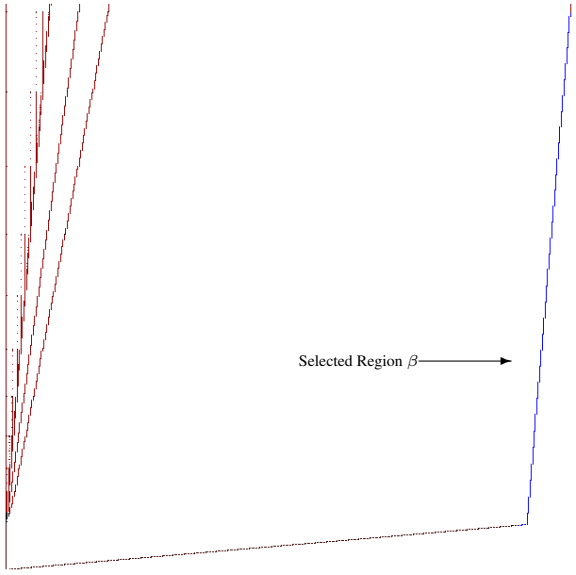
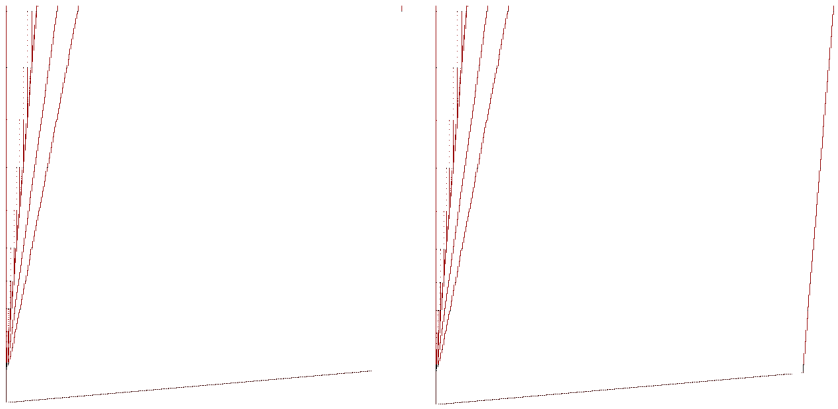


Figure 6.25: Selected Instructions for Dead-Hot Test



(a) Top 25% of Hot Code Region β (b) Bottom 75% of Hot Code Region β

Figure 6.26: 254.gap Region β

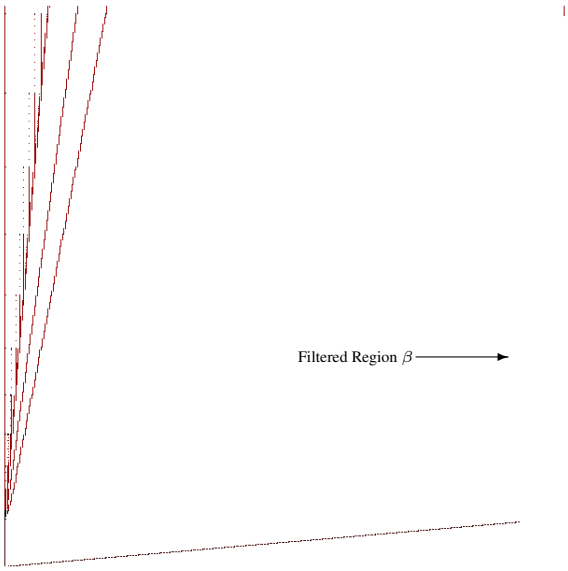


Figure 6.27: Dead-Hot Filtered Region

Chapter 7

Coarse-Grained Thread Level Parallelism

Instruction level parallelism (ILP) limitations have forced processor manufacturers to develop multi-core platforms with the expectation that programs will be able to exploit thread level parallelism (TLP). An effective, popular, and widely studied mechanism for automatically exploiting parallelism is to dynamically and speculatively thread groups of instructions [15, 20, 25, 64, 73, 79, 87, 90, 93, 97, 101]. Recent advances in this thread level speculation (TLS) can parallelize execute over 90% of some codes [64]. However, TLS predictor accuracy limits TLS to fine-grained or loop-level TLP [15, 64, 79, 90, 98].

This thesis explores potential coarse grain TLP that may be exploitable in conjunction with TLS and ILP techniques. In particular, the thesis examines the SPEC INT 2006 benchmark suite, looking for parallelism with a granularity of thousands of dynamic instructions, and is not restricted to loop-level TLP. Because the parallelism explored here is coarse grained, it may be able to work synergistically with techniques designed for a smaller granularity, such as TLS. Coarse-grained TLP is located using a dynamic trace visualization created by the ParaMeter tool [78]. This technique, which is discussed in Section 7.1, generates a visualization of program execution, called a DINxRDY (dynamic instruction number by ready time) plot. This plot visually shows potential coarse grain TLP as lines that overlap on the x-axis.

Potential parallel regions found within DINxRDY plots are further analyzed to expose dependence relationships. Inter-region dependence conflicts, discussed in Section 7.4, are found by dynamic dependence graph (DDG) slicing [4, 5, 30, 52]. Slicing DDGs that contain a large number of instructions (e.g. billions) can take weeks [5, 104]. To efficiently, and precisely, explore the dependency relationships between two regions of code this thesis extends the DDG *chop* to use the ZDD-compressed trace format. The dynamic chop [36, 54]

is often faster than an intersection of a forward and reverse slice and requires no loss of precision.

The thesis shows that on average, 7% of instructions may be extracted as coarse-grained parallelism, and in some cases, as much as 44% of instructions may be extracted as coarse-grained TLP.

7.1 TLP Visualization with ParaMeter

The ParaMeter tool [78], used for analysis and visualization in this thesis, uses instruction-level dynamic trace information for program visualization. Precise dynamic trace information can quickly grow in size and must be compressed or abstracted. ParaMeter uses zero suppressed binary decision diagrams to create a trace representation that provides compression, but is analyzable in its compressed form. Other tools, such as SD3 [50], use also use analyzable compressed forms, such as stride compression or hierarchical grammars [58], for dynamic trace representation. Stride compressors naturally find regions of parallelism that exist inside loops. This thesis presents a study of thread level parallelism that includes potential opportunities for parallelism without knowledge program structure, including loops.

ParaMeter generates plots of (DIN, RDY) as the primary form of visualization [78]. Dependence chains, or DDCs, form lines in the DINxRDY time plot. DDCs that overlap in the DINxRDY plot are targets for parallel thread extraction Figure 7.1 shows a sample (DIN, RDY) visualization of 175.vpr used in a case study of visualization of program parallelism [78]. Note the lines labeled α, β, and γ overlap on the x-axis, which represents the RDY time value. If the lines in the (DIN, RDY) plot correspond to distinct regions of code, and they overlap on the RDY axis, the source code has the potential to be parallelized. A single iteration of a reverse dynamic program slicing was used in this work to find the dependence relationship between α, β, and γ [78]. The regions α and β were found to be completely independent areas in the source code, and were threaded using standard pthreads. The resulting code executed all test benchmark inputs without error.

7.2 Region Selection

The human brain has a keen ability identify patterns in images [13]. Pattern recognition is used by ParaMeter to locate regions within DINxRDY visualizations that contain potential thread level parallelism,

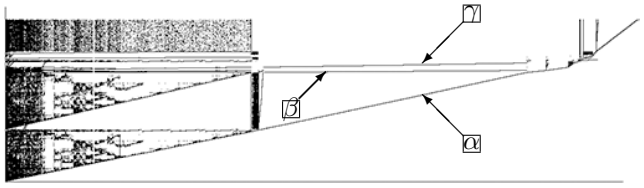


Figure 7.1: Overview DINxRDY Plot of 175.vpr

such as regions shown in Figure 7.1. The pattern that often corresponds to a *good* potential TLP region may be described informally as follows:

(1) The region should overlap another region on the RDY time axis

(2) At a given RDY time location, the region should not extend over the DIN axis

The first point reiterates the idea that two DDCs that overlap on the READY time axis can potentially be extracted as TLP. The second point addresses regions that contain a large amount of ILP. The instructions that form this ILP in a *bad* region often originates from many static instructions from an unrelated locations in the source code, and are difficult to compose as a thread.

The pattern may be quantified by examining the selected regions and the static code. Static instruction numbers, or SINs, connect the dynamic trace information to the program source. However, sets of DIN,SIN tuples also can characterize hard-to-parallelize regions. A *good* region can be clustered into nearly contiguous strides of static instructions. The number of resulting clusters should be small.

The description of a good region leaves room for interpretation. First, the distance between static instructions should be as close to contiguous as possible, with some margin for variable-length instructions. Second, a cluster is often part of a distinct section in the source code (such as a function), and a region that encompasses multiple functions may still be parallelizable.

Clusters for this work were found using a standard k-means clustering algorithm. A cluster contains a set of tuples static instructions, or SINs, and the DINs created by each SIN. A static instruction may be executed many times within a dynamic trace, thus the DIN,SIN set from a region selected from the DINxRDY

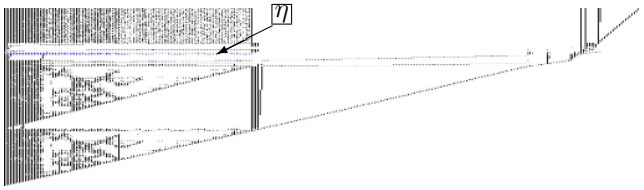
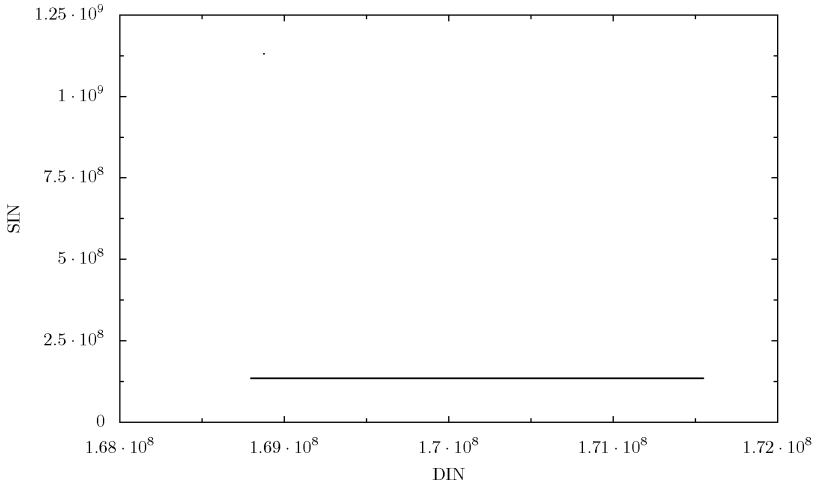


Figure 7.2: DINxRDY for Bad Region Selection (η)

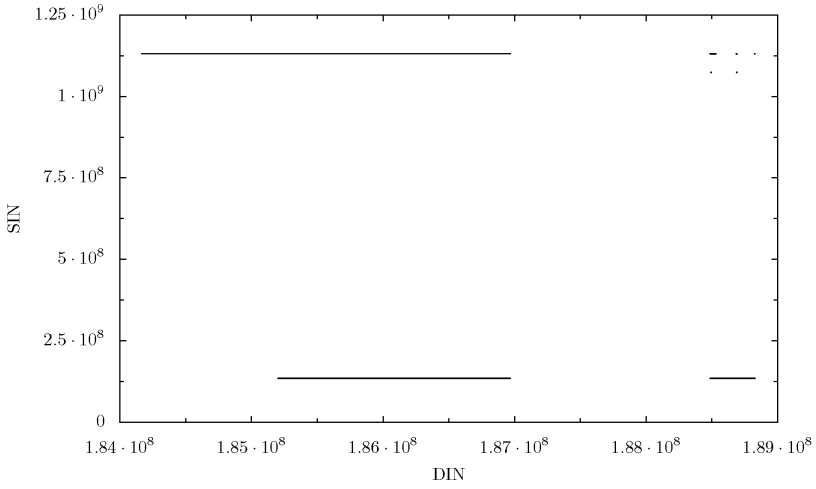
visualization. The RDY tuple member is removed from the selected set, and the resulting DIN values are intersected with a DINxSIN tuple set. The clusters are then formed from DIN,SIN tuples. If a DINxRDY selection requires a large number of clusters to meet this requirement, then the static source code for the region would be difficult to include in a thread. It is important to note that this technique only requires SIN values to be contiguous in a cluster, not all SIN values in a given region. This allows *good* regions to contain function calls and other forms of branching.

The regions α, β, and γ were found to be part of three distinct areas in source code [78]. Figure 7.2 presents an example of a selected region, region η, that does not contain instructions that would easily be extracted as TLP. The region β, was identified as easily extractable TLP in prior work [78], can be seen in the DINxSIN plot shown in Figure 7.3. Only a tiny point, in the upper left corner of the plot for β, deviates from the line extending across the bottom DIN axis. A DINxSIN plot for η, shown in 7.3, contains a number of lines that extend over the DIN axis. Each line may be a separate distinguishable section in the source code. Further, there are a number of points that do not form a line. Thus, composing the region η as a thread, for TLP, requires each instructions to be removed from their original location in the source code and placed in this new thread.

The regions selected from the SPEC 2006 INT benchmark suite can be found in Appendix B. The source code for the selected regions can be found in Appendix F. The source code in Appendix F has been abstracted to the function level due to the tens of thousands of lines of code, mostly C and C++, included in these regions. Some regions also contained thousands of functions. This number was reduced by creating an additional abstraction that truncates functions with similar letters at the beginning of the function name.

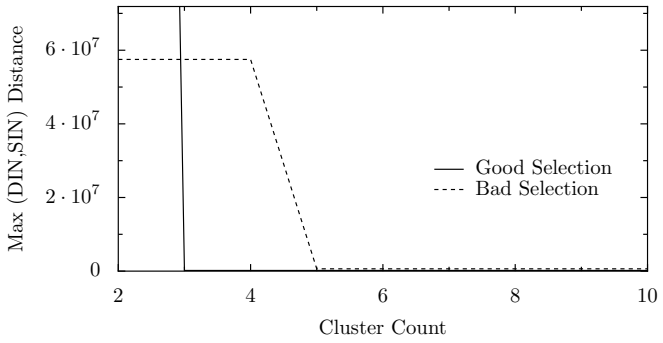


(a) 175.vpr Good Region Selection β

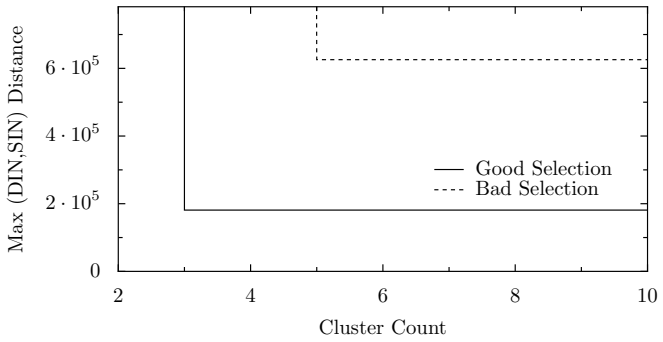


(b) 175.vpr Bad Region Selection η

Figure 7.3: DINxSIN for Selected Regions



(a) Region Clustering



(b) Region Clustering Zoomed

Figure 7.4: k-Means Clustering vs. Maximum DIN,SIN Distance

```
function iter_reverse_slice(e,I_d)
  s_old = 0// Empty set
  s := I_d
  s_total := s
  while s_full ≠ s_old
    s_old := s_full
    s := reverse_slice_bdd(e, s)
    s_total := s_full ∪ s
  return s_full
```

Figure 7.5: Computing a Reverse Slice to Convergence using BDDs.

For example, the functions *math::round* and math::abs may become *math::*. Functions that required this additional abstraction are truncated with ..., so our example would become *math::*.... Note that this truncation was determined not by the length of the function name, but the breadth of a trie containing the function name; for this thesis, the breadth limit was set at twenty.

7.3 ZDD Slicing

Dynamic program slicing has been used in prior work to find the source of observed software bugs [4, 5, 30], or to perform points-to analysis [62]. The BDD based slicing algorithm for ParaMeter is shown in Figure 5.1 [75]. The pseudo code in Figure 5.1 shows how to compute the reverse slice of an instruction with DIN d [75]. In this pseudo-code, e is the indicator function for the set of edges in the data dependence graph, I_d is the indicator function for the DIN d, and s is the indicator function for the reverse slice that is computed. The variables in s, as well as variables for indicator function I_d, are in the vector $\mathbf{d^1}$. The variables in e are $(\mathbf{d^1}, \mathbf{d^2})$. The function rename takes a function s and renames the variables from set $\mathbf{d^2}$ to the corresponding variables in set $\mathbf{d^1}$.

The algorithm in Figure 5.1, located in Chapter 5, can be extended to return the set of indicator functions for all variables in the reverse slice of d by iterating until a set s_{full} converges. The pseudo-code for this function is shown in Figure 7.5.

Program slicing with ZDDs requires modification to the pseudo-code in Figure 7.5 and Figure 5.1.

function reverse_slice_zdd(e,I_d)
  $s := I_d$
  $s = \mathrm{rename}(\mathbf{d^1}, \mathbf{d^2}, s)$
  $s = \mathrm{yDC}(s)$
  $e' := e \cap s$
  $s := \exists \mathbf{d^2}.e'$
  return s

Figure 7.6: Computing a Single Reverse Slice using ZDDs.

Slicing often requires the use of existential quantification, shown with the symbol $\exists$, to remove variables from a function represented by a ZDD or BDD. A missing variable in the BDD structure is interpreted as a boolean *don't care* value. A variable missing from a ZDD may be interpreted as a zero value or *don't care*. An additional step is taken to insert *don't care* values into a ZDD-based function to perform operations that intersect with *don't care* variables. The slicing algorithm shown in Figure 7.6 uses the function $\mathrm{yDC}(d)$ to insert *don't care* values in the y variable positions. This function, given the variable positions for the tuple (x, y) and an identity function d, will return the ZDD dot product [65] of the function d and the universal set of bits for tuple variable y. A similar function, $\mathrm{xDC(s)}$, is used for forward slicing.

The pseudo-code shown in Figure 7.7 performs a union of s with the result of function f until s converges. Note that comparison is a constant time operation with ZDDs, like BDDs [17].

Iteration of a dynamic dependence graph (DDG) slice until convergence can be $O(2^n)$ for both space and computation, where n is the number instructions being sliced [5, 91]. The worst case $O(2^n)$ computation assumes each slice instruction contains at most two subsets. ZDD-based slicing also requires $O(2^n)$ space. However, ZDD representation often provides adequate compression (better than 10x) [76]. The ZDD-based slice computation presented in Figure 7.7 computes the slice for both subsets of for a single slice iteration simultaneously.

```
function iter_slice_zdd(e,I_d,f, n)
  s_old = 0// Empty set
  count := 0
  s := I_d
  s_total := s
  while ((s_full ≠ s_old)∧
         ((n == 0) ∨ (count < n)))
    s_old := s_full
    s := f(e, s)
    s_total := s_full ∪ s
    count := count + 1
  return s_full
```

Figure 7.7: Fixed-Point Slice Computation with ZDDs.

function forward_edge_slice(e,e_d)
  $s := \exists \mathbf{d^2}.e_d$
  $s = \text{rename}(\mathbf{d^2}, \mathbf{d^1}, s)$
  $s = \text{xDC}(s)$
  $e_s := e \cap s$
  return e_s

Figure 7.8: Computing a Single Forward Slice using ZDDs.

7.4 ZDD Chopping

Unfortunately, performing a reverse slice of 1 billion dynamic instructions can take days; often the processes were terminated before completion. The survey presented in this work determines the dependence relationship of two or more sets of instructions. Therefore, for each instruction region, we can compute the slice of that region with respect to all other regions. This operation is known as a DDG *chop* [36, 54].

A ZDD chop is provided two sets of instructions d_1 and d_2 in the form of the identity functions I_{d1} and I_{d2} respectively. The set e contains the edges of the DDG in the form $(\mathbf{d^1}, \mathbf{d^2})$. The tuple $(\mathbf{d^1}, \mathbf{d^2})$ may labeled as (x, y), in this thesis. In ParaMeter, the set $(\mathbf{d^1}, \mathbf{d^2})$ is also called the (DIN, DIN) set [78]. The ZDD chop begins by generating the edge set during the forward slice. The new edge set, e_s, contains the edges in the DDG e that extend from the first instruction set to the end of the DDG. The pseudo-code for one iteration of this function is shown in Figure 7.8.

Figure 7.9 contains pseudo-code for producing a ZDD-based DDG chop. An initial edge set e_{d1} is constructed by intersecting the union of I_{d1} and the *don't care* for the y variable positions with the edge set e that contains the entire DDG for our trace. The iterator function is this called on the function $\text{forward\_edge\_slice}$ to produce the full forward slice of edge values from I_{d1}, which is stored in e_f. Finally, the iterator function is called again using the reverse slice function for the instruction set I_{d2} using the new edge set e_f. This processes is illustrated in Figure 7.10

function ZDD_Chop(e,I_{d1},I_{d2})
  $s := I_d$
  $e_{d1} := \mathrm{yDC}(I_{d1}) \cup e$
  $e_f := \mathrm{iter\_slice\_zdd}(e, e_{d1}, \mathrm{forward\_edge\_slice}, 0)$
  $s := \mathrm{iter\_slice\_zdd}(I_{d2}, e_f, \mathrm{reverse\_slice\_zdd}, 0)$
  return s

Figure 7.9: ZDD Chop

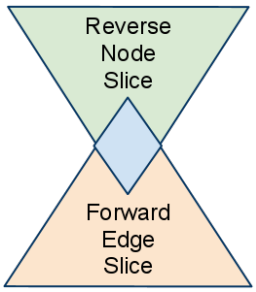


Figure 7.10: ZDD Chop Illustrated

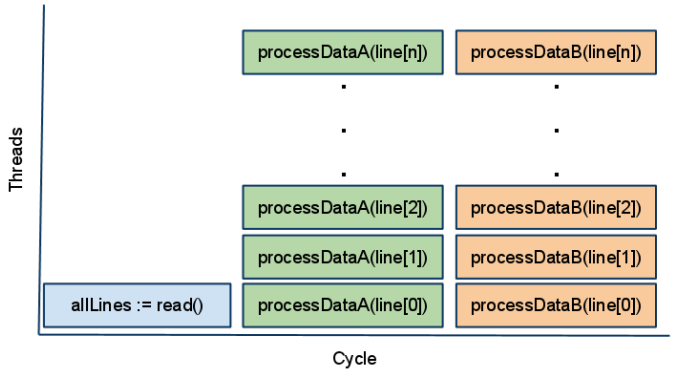


Figure 7.11: Fine-Grained from Figure 7.12

7.5 Potential Coarse-Grained Thread Level Parallelism in SPEC INT 2006

This study explores potential coarse-grained thread level parallelism in the SPEC 2006 INT benchmark suite. Traces were capped at one billion dynamic instructions. The benchmarks 400.perlbench, 403.gcc, 445.go, and 483.xalanchbmk *-O2* were traced completely. Traces used for this survey were collected using 64-bit x86 Intel Xeon processors.

7.5.1 Fine-Grained Harmony

The TLP located by the technique presented in this thesis should have a minimum impact on finer-grained TLP extraction techniques. Therefore, potential TLP detected by the techniques presented in this thesis will likely work with thread-level speculation or instruction-level parallelism.

Consider the example code presented in Figure 7.12. Lets assume that the functions $processDataA()$ and $processDataB()$ do not modify the program state to simplify the example. Each call to $processDataA()$ and $processDataB()$ is independent, therefore TLS could execute all iterations of the loop in $processDataA()$. It should also be possible to execute all iterations of $processDataA()$ as parallel threads with TLS. This execution is illustrate in Figure 7.11.

Note that Figure 7.12 also contains course-grained TLP. For example, assume the DINxRDY visualization, discussed in Section 7.1 can determine $processDataA()$ and $processDataB()$ do not depend on each other, but DINxRDY visualization is unable to determine each loop iteration is independent. Thus, DINxRDY

```
function potential_parallel_main()
  allLines := read()// Read Lines from File
  foreach line in allLines
    processDataA(line)
  foreach line in allLines
    processDataB(line)
```

Figure 7.12: Potential Parallel Code Example.

visualizations could locate the coarse-grained TLP, but not the fine-grained TLP. In Figure 7.13 illustrates the execution of coarse-grained TLP from the source code in Figure 7.12.

The course-grained parallelism and fine-grained parallelism can work independently to locate the parallel execution seen in Figure 7.13 and Figure 7.11, respectively. The combination of two techniques can extract greater amounts of TLP. Figure 7.14 shows the execution after extracting coarse-grained parallelism using the DINxRDY visualizations, and executing each thread in a system that contains TLS.

The Figure 7.15 contains the percent of parallelized instructions found in this work and the performance gained from a TLS system that uses perform control, data dependence, and data value speculation [48].

7.5.2 Compiler Influence

A compiler optimization pass, such as loop unrolling, constant propagation, or constant folding, can expose opportunities for parallelism. Compiler optimization may also remove potential TLP regions though dead-code elimination. This thesis examines TLP increases for programs compiled with gcc 4.2.4 using the *-O0* and *-O2* flags. The gcc documentation describes *-O0* as unoptimized code, and *-O2* as "nearly all supported optimizations that do not involve a space-speed trade-off."

A difference in potential coarse-grained TLP can be seen in the *-O0* and *-O2* for all benchmarks. This thesis will look further at the output found from 401.bzip2 benchmark. The bzip2 application is generally considered to be highly amenable to TLP optimization through manual high-level software changes [33]. In Figure 7.18 we can seen that a large amount of potential does TLP exists for in the DINxRDY plot from both compilation settings. However, Figure 7.18 does show a 12% difference with the *-O0* and *-O2* compiler

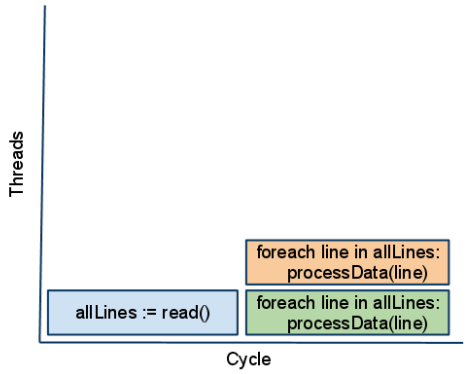


Figure 7.13: Fine-Grained from Figure 7.12

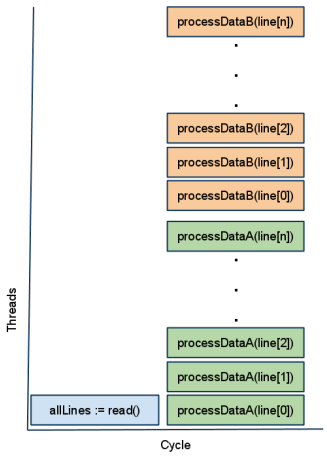


Figure 7.14: Coarse-and-Fine Grained from Figure 7.12

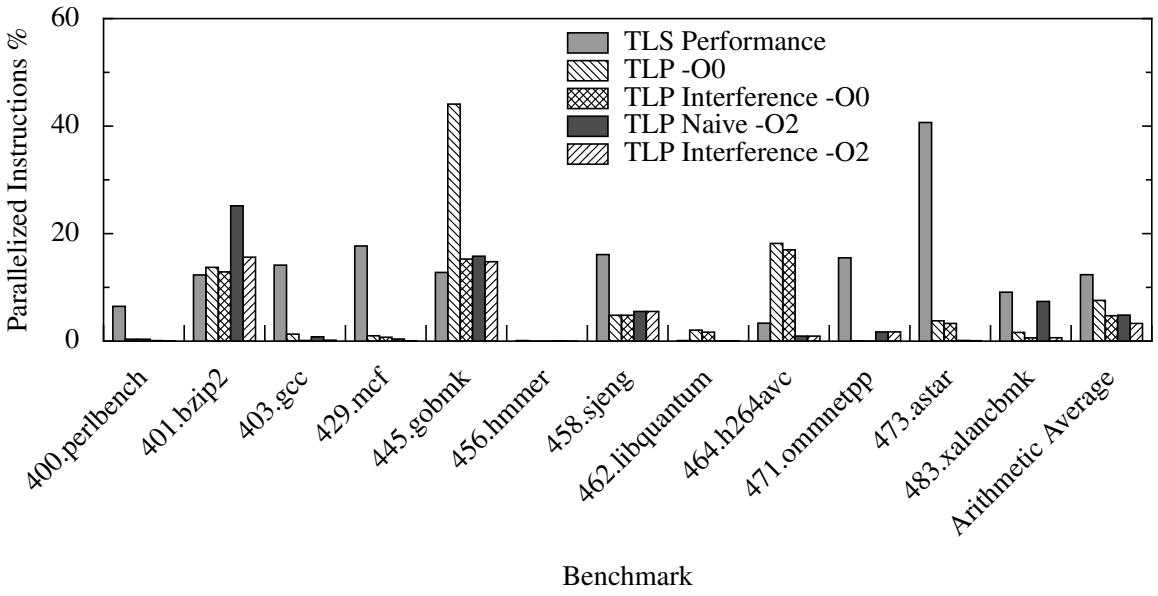
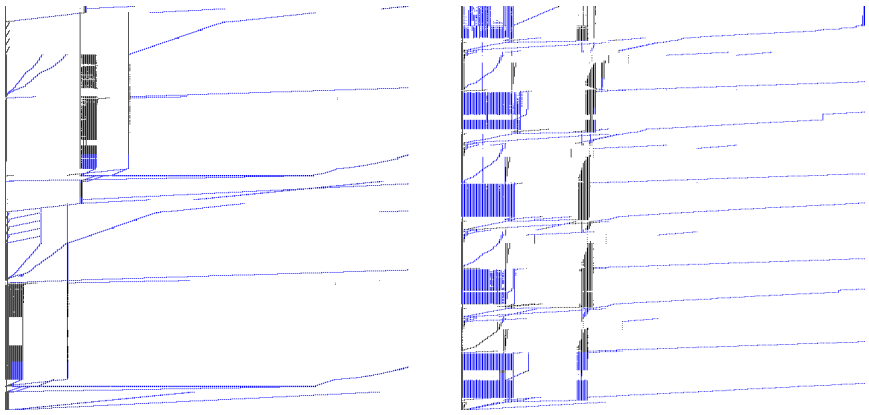


Figure 7.15: Coarse-Grained TLP and TLS [48]



(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure 7.16: 401.bzip2 with Selected Regions

settings for 401.bzip2.

The regions selected as potential TLP can be seen in Figure 7.16. The parallel regions in 401.bzip for both gcc settings were traced back to source code. Examination of the source code responsible for TLP found that, while some parallel instructions differed between optimization settings, all enclosing functions were identical. A list of the functions that contain TLP for both *-O0* and *-O2* is shown in Figure 7.17. The *-O2* compiler optimization setting reduced the number of instructions in each thread, thus reducing the benefit of parallel execution.

7.5.3 Summary of Potential Coarse-Grained TLP

Figure 7.18 shows the percent of instructions identified as potential coarse-grained parallelism, given optimistic speculative execution. Optimistic speculative execution ignores all dynamic dependencies. Figures 7.18 and Figure 7.20 show the percentage of dynamic instructions that may potentially execute in the same cycle as another instruction. A potential parallel instruction is only allowed to be counted a single time. The Figure 7.19 shows the maximum number of potential parallel regions that overlap on the READY axis in the DINxRDY plot. Potential parallel instructions are allowed to be counted twice in Figure 7.19; the optimistic algorithm would otherwise remove an entire region from the thread count, thus artificially restricting the optimistic thread count.

```
BZ2_blockSort
BZ2_bzCompress
BZ2_bzWrite
BZ2_compressBlock
BZ2_hbAssignCodes
BZ2_hbMakeCodeLengths
add_pair_to_block
bsPutUInt32
bsW
compressStream
copy_input_until_stop
copy_output_until_stop
generateMTFValues
handle_compress
mainGtU
mainQSort3
mainSimpleSort
mainSort
makeMaps_e
mmed3
myfeof
prepare_new_block
sendMTFValues
spec_fread
spec_fwrite
spec_getc
spec_ungetc
```

Figure 7.17: 401.bzip2 -O0 and -O2 Functions with TLP

It should also be noted that finding the thread counts in Figures 7.19 and 7.21 can be difficult to calculate. Fore example, it may be possible to delay the execution of a potential thread to resolve dependencies, and thus allow for a greater maximum thread count. Therefore, Figures 7.19 and 7.21 should be considered approximations of the true maximum thread count.

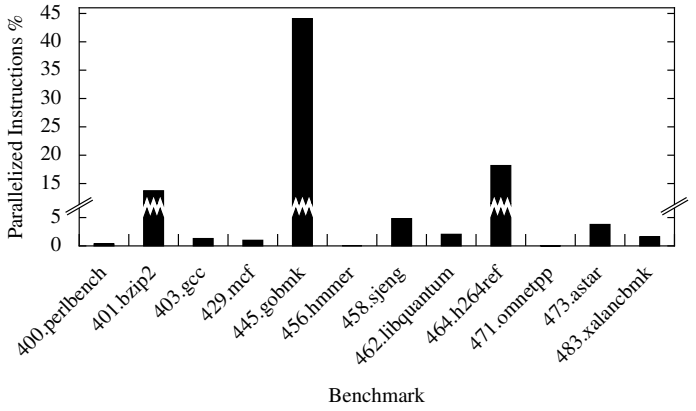
The ZDD-chop is used to calculate interdependent instructions that exist in two or more selected regions. Figure 7.21 was calculated using the number of overlapping selected regions on the READY time axis that contain potential parallel instructions. The results in Figure 7.20 and Figure 7.21 remove all interdependent instructions found in the dynamic trace, and also do not allow duplicate parallel instructions.

This work further explores the potential coarse-grained TLP found in the benchmarks 445.gobmk, 400.perlbench, and 462.libquantum. The DINxRDY visualizations and potential coarse-grained TLP for 456.hmmer and 471.omnetpp have properties that closely resemble 400.perlbench. The benchmarks 401.bzip2, 464.h264ref, and 445.gobmk produce visualizations and potential coarse-grained TLP with similar traits. Visualizations and potential coarse-grained TLP generated by 458.sjeng and 473.astar have features that closely resemble 462.libquantum.

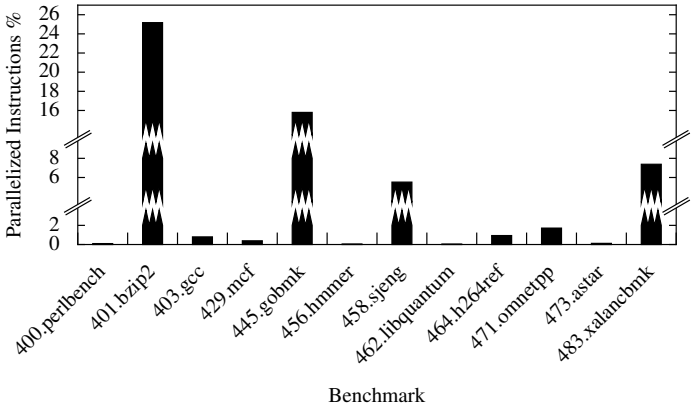
7.5.4 Potential Coarse-Grained TLP in 400.perlbench

Less than 1% of the dynamic instructions in the benchmarks 400.perlbench, 456.hmmer, and 471.omnetpp were identified as potential coarse-grained TLP. The benchmark 400.perlbench contains two regions with potential TLP, and each region contains two potential threads. The Figure 7.23 shows the DINxRDY plot of 400.perlbench *-O0* and *-O2* with potential TLP highlighted. The DINxRDY plots in Figure B.1 show few dynamic dependence chains that overlap on the horizontal RDY axis.

Figure 7.23 corresponds to source code that performs file I/O, lexing, parsing, and other functions in the Perl interpreter. Each region for 400.perlbench contain threads with similar source code. For example, the first thread in region 1 and region 2 contain instructions for lexing, parsing and IO (*Perl_yylex*, *Perl_yyparse*). The second thread in region 1 and 2 both contain functions for the Perl-to-C interpreter(*S_new_xpvbm*). Thus, it may be possible for lexing and parsing operations to work in parallel with other instructions in the Perl interpreter, perhaps as a software pipelines [7, 32]. However, the two regions from 400.perlbench likely

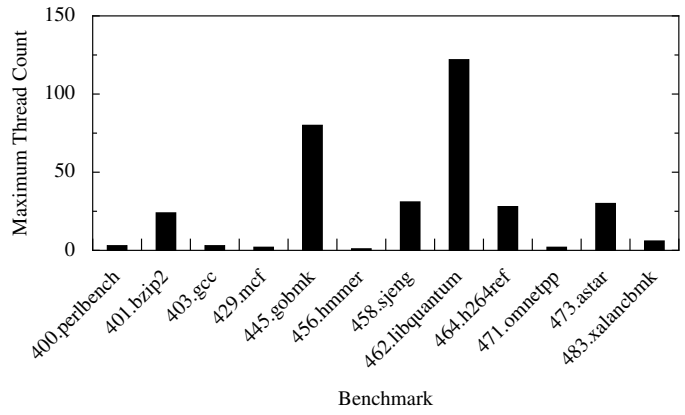


(a) GCC -O0 Optimizations

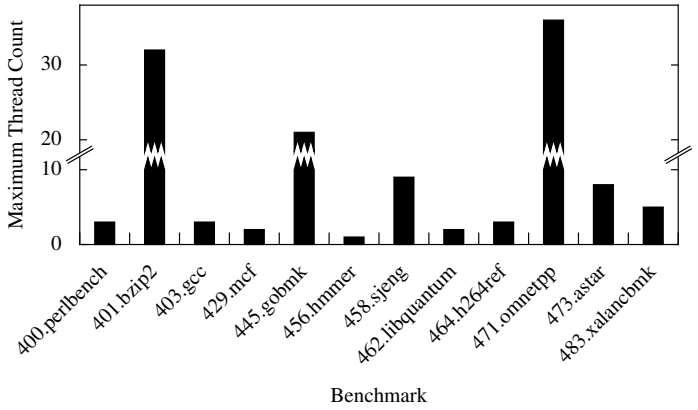


(b) GCC -O2 Optimizations

Figure 7.18: Potential Coarse-Grained TLP

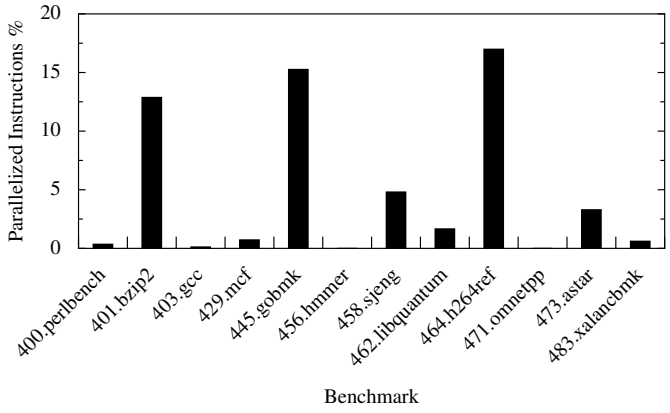


(a) GCC -O0 Optimizations

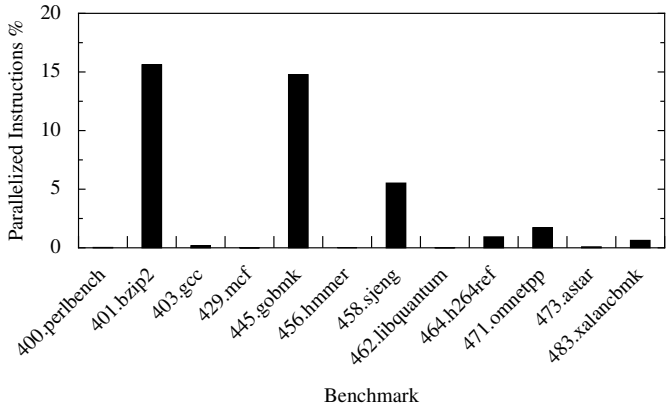


(b) GCC -O2 Optimizations

Figure 7.19: Optimistic Thread Count

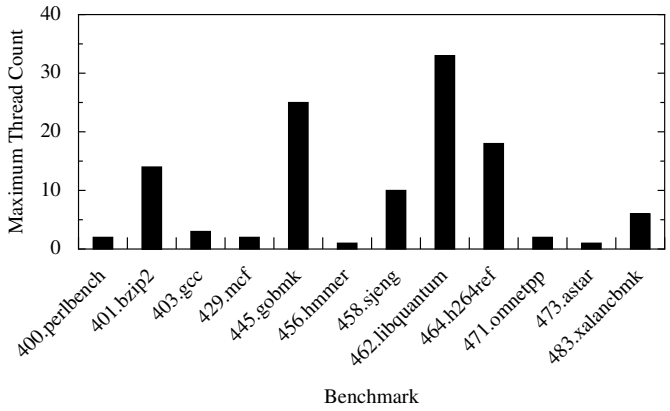


(a) GCC -O0 Optimizations

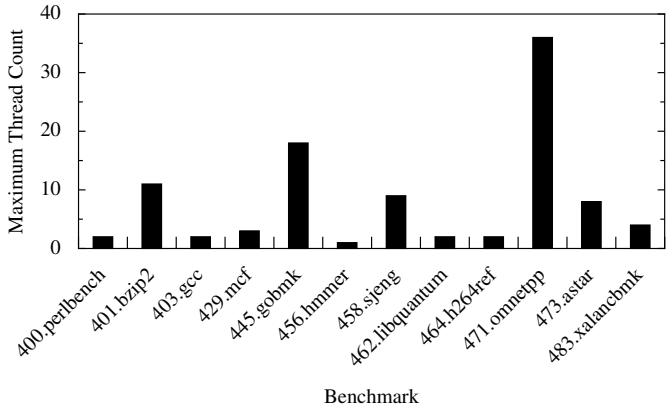


(b) GCC -O2 Optimizations

Figure 7.20: Potential TLP W/0 Interdependent Instructions



(a) GCC -O0 Optimizations



(b) GCC -O2 Optimizations

Figure 7.21: Maximum Threads W/0 Interdependent Instructions

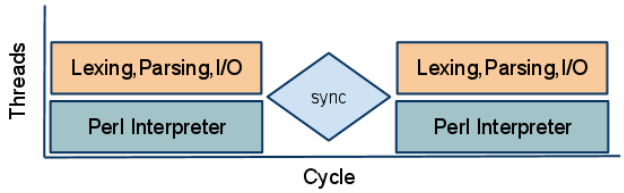
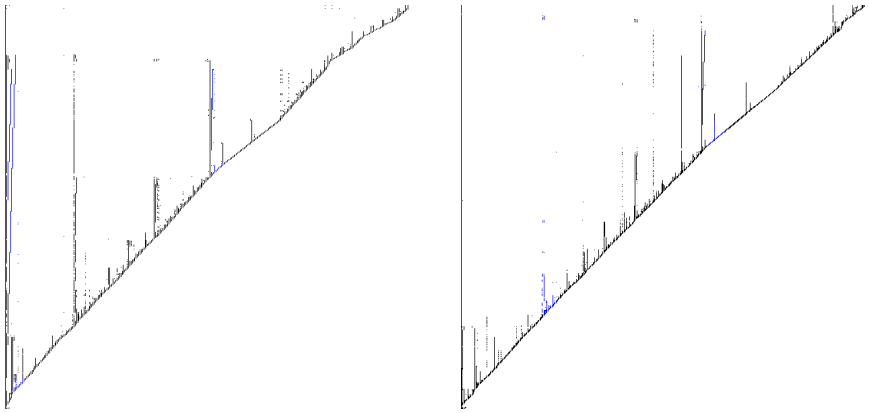


Figure 7.22: 400.perlbench TLP with Synchronization



(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure 7.23: 400.perlbench with Selected Potential TLP

must execute in sequence. Therefore, 400.perlbench may contain two stages with potential coarse-grained TLP, and synchronization between stages. Figure 7.22 illustrates the possible interaction between potential coarse-grained TLP in 400.perlbench.

The Figure 7.24 and Figure 7.25 contain an abridged source code list for potential threads, with parallel threads grouped into regions. For example, Region 1 contains two threads that could potentially execute in parallel.

The DINxRDY visualization and source code for the benchmarks 456.hmmer and 471.omnetpp have properties similar to 400.perlbench.

7.5.5 Potential Coarse-Grained TLP in 445.gobmk

The benchmark 445.gobmk contains the greatest percentage of potential coarse-grained parallelism. Figures 7.18 and 7.20 show that TLP in 445.gobmk is reduced by 29% when inter-region instruction dependencies are removed. Inter-region dependent instructions depend on a result from an instruction in another region. The DINxRDY plot for *-O0* in Figure 7.28 contains 123 selected dynamic dependence chains that overlap on the horizontal RDY axis for the *-O0* binary, resulting in over 44% of the dynamic instructions are candidates for coarse-grained TLP. Note that many selected regions in 7.28 also contain instructions stacked vertically; instructions that extend vertically along the DIN axis can potentially be parallelized by ILP extraction or fine-grained TLS.

The source code responsible for the potential TLP shown in Figure 7.28 perform a variety of functions that play the game Go. These include *do_trymove*, *defend1*, *order_moves*, and *is_suicide*. Each potential thread contains similar source code, which implies multiple Go moves can be tested at the same time.

Potential coarse-grained TLP in 445.gobmk is reduced by 29% when inter-region instruction dependencies are removed, which is the largest reduction of all surveyed benchmarks. Instructions that depend on the result of an instruction in another parallel region include *special_attack3*, *find_defense*, and *is_suicide*. These instructions, and some others, likely require the results from a previous Go move. The highlighted portions of Figure 7.29 represent instructions that do not contain an inter-region dependency. Note that few regions shown in *-O0* in Figure 7.28 do not depend on a result from an instruction in another region.

PerlIOBuf
Perl_append
Perl_allocmy
Perl_ck_concat
Perl_convert
Perl_cv_undef
Perl_bind_match
Perl_block_start
Perl_gv_init
Perl_intro_my
Perl_hv_iterinit
Perl_keyword
Perl_my_attrs
Perl_mod
Perl_listkids
Perl_localize
Perl_oopsAV
Perl_peep
Perl_scalar
Perl_scope
Perl_sv_grow
Perl_yyparse
Perl_yylex
Perl_jmaybe
S_call_body
S_cl_init
S_gv_init_sv
S_force_word
S_incline
S_intuit_more
S_hsplit
S_hv_magic_check
S_my_kid
S_more_xpvbm
S_modkids
S_new_xiv
S_pad_findlex

(a) Thread 1

S_new_xpvbm

(b) Thread 2

Figure 7.24: 400.perlbench TLP Source Code Region 1

PerlIOBase
Perl_allocmy
Perl_av_push
Perl_ck_fun
Perl_convert
Perl_block_gimme
Perl_grok_hex
Perl_gv_init
Perl_hv_iterinit
Perl_keyword
Perl_free_tmps
Perl_my_attrs
Perl_mg_magical
Perl_mod
Perl_listkids
Perl_localize
Perl_pad_new
Perl_peep
Perl_scalar
Perl_sv_grow
Perl_utilize
Perl_yyparse
Perl_yylex
S_call_body
S_dopoptosub
S_gv_init_sv
S_filter_gets
S_force_version
S_hv_fetch_common
S_my_kid
S_more_sv
S_new_xiv
S_scan_word
S_sublex_push

(a) Thread 1

Perl_append_elem
Perl_append_list
Perl_block_start
Perl_block_end
Perl_gv_init
Perl_intro_my
Perl_leave_scope
Perl_newOP
Perl_newFOROP
Perl_pregcomp
Perl_pmruntime
Perl_save_I32
Perl_yyparse
Perl_yylex
S_lop
S_new_xpvav
S_new_xpvcv

(b) Thread 2

Figure 7.25: 400.perlbench TLP Source Code Region 2

```
toke.c: 218-227 492-794 906-1404 1626-1743
        1864-2599 2784-2828 2931-2985 3116-3196
        3304-4107 4248-4388 4561-4717 4895-4897
        5028-5121 5276-5673 5818-5958 6342-6350
        6480-6588 6795-6951 8708-8737 8858-9171
        9639-10313 10421-10447

pad.c:  139-1181

perl.c: 2209-2382 4743-4850

perlio.c: 1599-1802 3776-3892

hv.c: 41-108 321-813 1064-1323 1649-1665
      1931-1933 2079-2133

op.c: 211-724 863-1046 1167-1528 1693-2385
      2710-2951 3203-3257 3402-3658 3998-4522
      4746-5291 5485-5505 5636-5792 6176-6220
      6383-6652 7009-7015

pp_hot.c: 44-329 2341-2396 2595-2695 2835-2996

sv.c: 633-654 785-1189 1294-1721 3254-3263
      3635-3811 3973-4176 4371-4384 4535-4807
      5045-5359 6105-6457 6744-6844 7040-7056

mg.c: 116-129 354-393

gv.c: 49-174 669-927 1115-1116 1258-1275

util.c: 72-165 375-431 804-817

av.c: 63-367 491-550 679-714

numeric.c: 32-46 515-747

perly.c: 1467-2521

run.c: 37-43

regcomp.c: 481-554 687-1070 1361-1379 1579-2173
           2430-2922 3096-3357 4351-4548 5090-5101

scope.c: 112-277 380-556 663-722 822-897 1019-1061
```

(a) Thread 1

```
sv.c: 1153-1158
```

(b) Thread 2

Figure 7.26: 400.perlbench TLP Source Code Lines Region 1

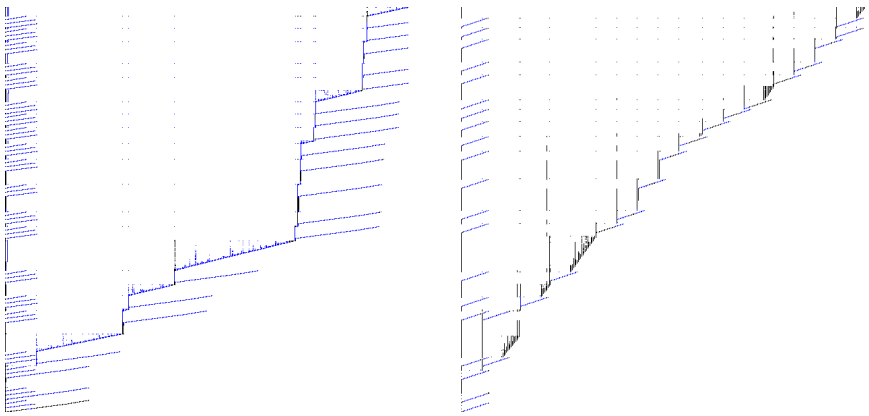
```
toke.c: 492-1526 1671-1710 2094-2599 2784-2828
        2964-2985 3116-3132 3373-3423 3679-3906
        4648-4662 5209-5423 5604-5853 8708-8946
        9047-9048 9639-10313 10421-10439

pad.c: 139-547 920-1181

perl.c: 2209-2382 4743-4850

perlio.c: 1575-1802 2050-2084 2487-2509
          3521-3652 3776-3892 4793-4801

hv.c: 41-108 321-408 573-794 1305-1323 1649-1665
      2079-2133

op.c: 211-722 875-1046 1210-1229 1340-1528
      1693-1796 1984-2385 2905-3091 3203-3483
      3998-4051 4175-4522 4826-4838 5108-5291
      5485-5505 5790-5899 6391-6537 7009-7015

pp_hot.c: 44-92 2341-2396 2595-2695 2835-2996

sv.c: 275-320 633-642 785-1021 1294-1721
      3254-3263 3635-3811 3973-4176
      4371-4384 4490-4807 5045-5359
      6105-6457 6744-6844 7050-7055

pp.c: 46-50

mg.c: 116-129

gv.c: 49-174 669-899 1115-1116 1258-1275

util.c: 72-165 804-817

av.c: 63-367 491-550 679-714

numeric.c: 263-350 515-747

perly.c: 1467-1677 1818-2064 2223-2521

run.c: 37-43

scope.c: 112-277 381-556 663-722
         822-897 1019-1061

pp_ctl.c: 1174-1230 2993-3079 3361-3362
```

(a) Thread 1

```
toke.c: 710 4649-5030

pad.c: 888-943

scope.c: 419-423 687-697

op.c: 1892-1917 2186-2330
      2754-2811 3424-3483 3767-3952

sv.c: 923-974

regcomp.c: 1748 2134

gv.c: 107-158

util.c: 72-101

perly.c: 1467-2521
```

(b) Thread 2

Figure 7.27: 400.perlbench TLP Source Code Lines Region 2



(a) GCC -O0 DINxRDY

(b) GCC -O2 DINxRDY

Figure 7.28: 445.gobmk with Selected Potential TLP

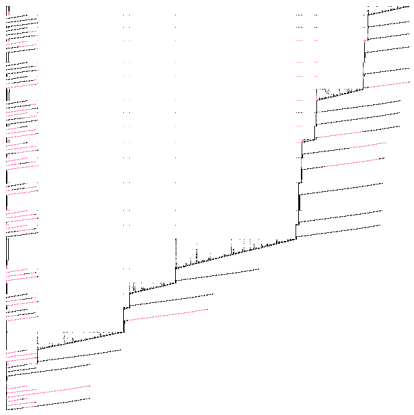


Figure 7.29: 445.gobmk Inter-Region Dependencies

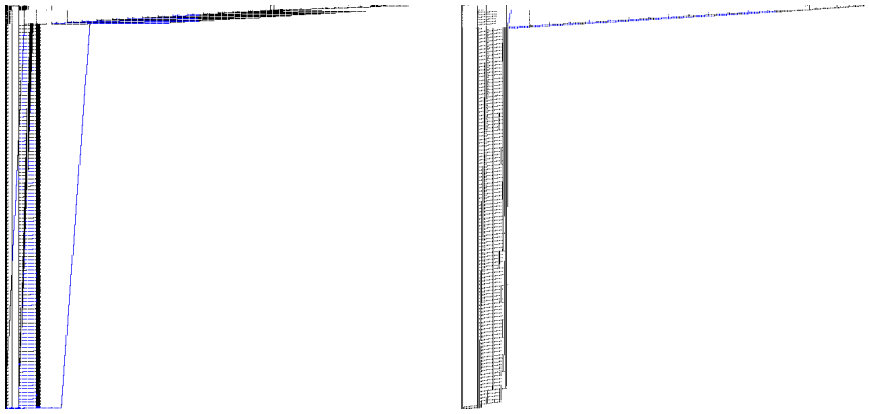
The benchmarks 401.bzip2 and 464.h264ref also contain a large percentage of coarse-grained TLP. The DINxRDY plots for 401.bzip2 and 464.h264ref have properties similar to 445.gobmk.

7.5.6 Potential Coarse-Grained TLP in 462.libquantum

The benchmark 462.libquantum is generally regarded as highly amenable to manual explicit TLP extraction [34]. However, as shown in Figure 7.15, most TLS systems extract only modest parallelism from 462.libquantum [48]. The results from the coarse-grained analysis performed in this work also show a modest percentage of potential TLP from 462.libquantum. Figure 7.30 contain the DINxRDY visualizations for *-O0* and *-O2* gcc optimization levels.

The source code responsible for the potential TLP shown in Figure 7.30 perform a number of scientific operations. Much like the benchmark 445.gobmk, the threads in each region of 462.libquantum are nearly identical. Figure 7.31 contains the source code of the first two threads in the first region.

Between 1% and 10% of the instructions in the benchmarks 458.sjeng, 462.libquantum and 473.astar are identified as potential coarse-grained TLP. The DINxRDY plots and source code for 458.sjeng and 473.astar have properties similar to 462.libquantum.



(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure 7.30: 462.libquantum with Selected Potential TLP

quantum_cnot	quantum_cnot
quantum_decohere	quantum_decohere
quantum_objcode_put	quantum_objcode_put
quantum_qec_get_status	quantum_qec_get_status
quantum_sigma_x	quantum_sigma_x
quantum_swaptheleads	quantum_swaptheleads
quantum_toffoli	quantum_toffoli
add_mod_n	add_mod_n
addn	addn
madd	madd
muxha	muxha
muxfa	muxfa
muln_inv	muln_inv
test_sum	test_sum

(a) Thread 1 (b) Thread 2

Figure 7.31: 462.libquantum TLP Source Code Lines Region 1

Chapter 8

Future Work

ZDD-compressed traces provide analyzable compression with reasonable creation time [76]. ZDD-compressed traces are highly adaptable to new analyses. This thesis begins to explore the use of dynamic visualization and analysis, but leaves a framework for further analysis and visualizations. This section highlights ideas that for future research.

8.1 Visualization

The visualization algorithm used in this thesis generates a simple, two dimensional representation of program execution [78]. The addition of coloring allows for hot-code region visualization, but the overall presentation does not provide the user with enough information to examine source code, or do simultaneous hot-code and dependence visualization. This image could be extended into three dimensions and allow the user to perform simultaneous static and dynamic program analyses. The combination of static and dynamic analysis has been explored by prior works with some success [27, 63, 70].

8.2 Dynamic Dependence Chain Classification

In addition to the standard thread-level parallelism discussed in this thesis, it may be possible to analyze a dependence chain and find other potential performance optimizations, such as software pipelines [7, 26, 46]. The ParaMeter tool can locate regions amenable to software pipelines [78], but such regions are not uniquely identified.

Chapter 9

What Just Happened?

This thesis demonstrated: (1) Zero-Suppressed Binary Decision Diagrams (ZDDs) enables many analyses to scale; (2) ZDD creation is practical for traces of a billion instructions for a variety of benchmarks; and, (3) ZDD-based analyses can reveal a number of performance opportunities that exist in sequential program traces. This chapter restates the motivation for this work, as well as summarizes the contributions of this dissertation.

9.1 Dynamic Trace Compression

Dynamic trace analysis has been used in prior work for performance tuning and hardware debugging [102]. Unfortunately, trace files can easily grow to terabytes in size depending on the information collected and the duration of traced execution. Large dynamic trace sizes (e.g. 1 billion instructions or more) can make analysis and visualization intractable.

Ideally, a compressed trace format should allow analyses to operate directly on the compressed representation with complexity that is a function of *compressed* trace size. Then, if large portions of compressed traces fit in memory, global analyses and interactive visualization are possible. Larus *et. al.* propose such a technique for *whole program path* analysis [58]. The technique uses the Sequitur compression method and works well for finding sequence matches in program execution. It does not permit direct application of data-centric analyses (e.g., trace slicing) that are of interest to system designers and programmers. Recent work on stride compression techniques addresses this issue by forming hierarchies based on accessed memory regions [50], but is limited to analyses based on loop-level dependence.

This thesis first explored reduced, ordered, binary decision diagram (ROBDDs) [17], originally developed for hardware verification, as a trace representation for dynamic program analysis. BDDs can provide compression for large sets of data whose size would otherwise make analysis intractable. When BDDs are used for the analysis of large program traces [75,78,105], the size of dynamic program traces can be reduced by up to 60x when encoded as a BDD [75]. Further, this compressed representation can be analyzed without decompression, with algorithmic complexity that is a function of the compressed size [75]. Thus, trace-encoded BDDs provide a solution to the dynamic trace size issue by representing trace information in a compressed, yet analyzable, structure.

9.2 Dynamic Trace Analysis at Scale

The SPEC 2006 INT benchmark suite, which is both large and diverse, requires nontrivial analyses to scale gracefully. Unfortunately, encoding large traces as BDDs can be time consuming, requiring hours to days to complete [77]. This, in turn, makes tools that use BDD-based representations less practical. Prior applications of BDDs depend on three methods to mitigate BDD creation time: (1) search for a variable order that allows for fast BDD creation; (2) tune the tables and caching systems used in many BDD packages; and, (3) encode an abstraction of the original data set, instead of the raw data. This thesis showed that, without any data abstraction, ZDD-based SPEC INT 2000 benchmark traces are 25% smaller than BDD-based traces. Furthermore, with the modification to a ZDD caching structure, ZDD-based trace compression results in a creation time that can be $9\times$ faster than BDD creation time for the same benchmark.

9.3 Dynamic Dependency Graph Slicing and Chopping

Slicing algorithms have been able to efficiently traverse the dynamic dependence graph (DDG) of dynamic trace data when encoded as BDDs [78]. Some algorithms, such as points-to analysis, have be adapted for ZDDs [62]. However, ZDD-based DDG slicing algorithms have yet to be explored beyond the most straight-forward translation from BDDs [76]. In Chapter 5, this dissertation explored two potential techniques for performing dynamic slicing with ZDD-encoded data sets.

The first method uses ZDD set intersection to identify dependency relationships that should be included

in the slice. The second technique uses the If-Then-Else (ITE) operator to identify common dependencies. Slices using intersection and slices using ITE were tested on benchmarks from the SPEC 2006 integer benchmark suite. Despite mixed results from these tests, the method based on set intersection showed good performance over most cases. The results from these tests were later incorporated into other analysis algorithms, such as dynamic dependence graph chopping used in TLP exploration and irrelevant instruction elimination.

9.4 Coarse-Grained Thread Level Parallelism

Instruction level parallelism (ILP) limitations have forced processor manufacturers to develop multi-core platforms with the expectation that parallel execution can be found at the thread level. This thesis examines the SPEC INT 2006 benchmark suite, looking for parallelism with a granularity of thousands of dynamic instructions, and is not restricted to loop-level thread level parallelism (TLP). Coarse-grained TLP is located using a dynamic trace visualization created by the ParaMeter tool [78]. This technique generates a visualization of program execution, called a DINxRDY (dynamic instruction number by ready time) plot. This plot visually shows potential coarse grain TLP as lines that overlap on the x-axis.

Potential parallel regions found within DINxRDY plots are further analyzed to expose dependence relationships. Inter-region dependence conflicts are found by dynamic dependence graph (DDG) slicing [4, 5, 30, 52]. Slicing DDGs that contain a large number of instructions (e.g. billions) can take weeks [5, 104]. To efficiently, and precisely, explore the dependency relationships between two regions of code this thesis extends the DDG *chop* to use the ZDD-compressed trace format. The dynamic chop [36, 54] is often faster than an intersection of a forward and reverse slice and requires no loss of precision.

The thesis shows that on average, 7% of instructions may be extracted as coarse-grained parallelism, and in some cases, as much as 44% of instructions may be extracted as coarse-grained TLP.

9.5 Irrelevant Instruction Elimination

Irrelevant component elimination has been adopted in prior work to simplify abstractions for static analysis [23]. This thesis uses irrelevant component elimination with ZDD-encoded precise dynamic

instruction dependencies. ZDD-based irrelevant instruction dependency elimination is designed to iterative irrelevant code calculation until convergence. However, irrelevant instruction dependency elimination can take days to complete for traces with long dynamic dependency chains. Empirical data presented in this thesis shows that, for all benchmarks in SPEC 2006 INT, the irrelevant instruction elimination algorithm reaches a steady state. In this state, the number of instruction dependencies removed per slice iteration oscillates, but will not monotonically decrease until the analysis converges. Thus, it is possible to approximate the number of instructions removed by iterating irrelevant instruction elimination until oscillation is detected.

ZDD-based irrelevant instruction dependency elimination was used to assist the study of TLP by filtering irrelevant points from program visualizations. Irrelevant instruction dependency elimination also can be used for code optimization or compiler evaluation. Chapter 6 contained a survey of irrelevant code removal from the SPEC 2006 INT benchmarks using both *-O0* and *-O2* optimization levels in the *gcc* compiler.

9.6 Hot-Code Visualization

In addition to the optimization analyzes presented in this thesis, which include thread-level parallel region location and irrelevant instruction elimination, a hot-code visualization algorithm was created to focus optimization efforts on the most frequently executed regions of code.

Hot-code analysis captures static instruction execution frequency and counts the frequency of execution. The static hot code information is combined with a mapping from $dynamic\ instruction \rightarrow static\ instructions$ to create a new relation from each dynamic instruction to hot-code value.

9.7 Contributions

Traces from a variety of applications need be analyzed to demonstrate the efficacy of DD-based dynamic trace analysis. The results in this thesis show that ZDD-based trace compression results in 25% smaller representation compared to BDD-based traces. Further, ZDDs have a smaller working set, thus the ZDD creation package can tuned to cache the working set of the trace-ZDD during creation. This reduces the number of garbage collection operations and removal of useful dead nodes. This reduces ZDD creation by up to $9\times$.

Hot code analysis can tell developers where to focus optimization and parallelization efforts. In addition to the optimization analyzes presented in this thesis, which include coarse-grained thread level parallel region location and irrelevant instruction elimination, a hot-code visualization algorithm was created to focus optimization efforts on the most frequently executed regions of code.

The results from a survey of irrelevant instructions in the SPEC 2006 INT benchmark shows that over 50% of instruction dependencies do not produce a value or reach the end of the program trace. It is possible for an instruction to be relevant to program execution but not meet the specified requirements. Therefore, this thesis also presents results comparing the irrelevant dependence counts from both the *-O0* and *-O2* compiler settings. The irrelevant dependency count from the *-O0*, or non-optimized, compiler setting provides a worst-case value to normalize further comparison operations. Furthermore, this technique can test the effectiveness of static compiler optimizations, as well as locate potential irrelevant instruction streams.

Finally, this thesis explores potential coarse grain TLP that may be exploitable in conjunction with TLS and ILP techniques. In particular, the thesis examines the SPEC INT 2006 benchmark suite, looking for parallelism with a granularity of thousands of dynamic instructions, and is not restricted to loop-level TLP. The survey presented in Chapter 7 found, on average, 7% of instructions may be extracted as course-grained parallelism, and for the benchmark 445.gobmk, 44% of instructions may be extracted as coarse-grained TLP.

The primary contributions of this thesis include:

(1) A ZDD-based trace compression algorithm with tuning for large traces, resulting in a 25% reduction in BDD size and a $9\times$ reduction in trace compression time.

(2) A ZDD-based iterative analysis for locating irrelevant instruction dependencies

(3) A method to quickly explore coarse-grained parallelism in serial applications.

(4) A ZDD-based dependence graph chopping algorithm need for the aforementioned method.

(5) A survey of coarse-grained thread level parallelism in the SPEC INT 2006 benchmarks that shows up to 44% of instructions may be extractable as coarse grain TLP.

Bibliography

[1] Dual core era begins, PC makers start selling Intel-based PCs: Intel dual-core processor-powered PC systems first to market. Technical report, Intel Corporation, April 2005.

[2] The first step in the multi-core revolution. Technical report, Intel Corporation, April 2005.

[3] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Transactional programming in a multi-core environment. In Katherine A. Yelick and John M. Mellor-Crummey, editors, Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2007, San Jose, California, USA, March 14-17, 2007, page 272. ACM, 2007.

[4] Hiralal Agrawal. Towards automatic debugging of computer programs. PhD thesis, West Lafayette, IN, USA, 1992.

[5] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation, pages 246–256, New York, NY, USA, 1990. ACM.

[6] S. B. Akers. Binary decision diagrams. IEEE Trans. Comput., 27(6):509–516, 1978.

[7] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. ACM Comput. Surv., 27(3):367–432, 1995.

[8] F. E. Allen and J. Cocke. A program data flow analysis procedure. Communications of the ACM, 19:137–147, March 1976.

[9] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/, 2010.

[10] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. SIGPLAN Not., 32(5):85–96, 1997.

[11] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In Proceedings of the 19th International Symposium on Computer Architecture (ISCA-19), May 1992.

[12] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM.

[13] Irving Biederman. Recognition by components: A theory of human image understanding. Psychological Review, pages 115–147, 1987.

[14] Robert K. Brayton, Gary D. Hachtel, Alberto L. Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen A. Edwards, Sunil P. Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy, and Tiziano Villa. Vis: A system for verification and synthesis. In CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification, pages 428–432, London, UK, 1996. Springer-Verlag.

[15] Matthew Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David August. Revisiting the sequential programming model for multi-core. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, pages 69–84, Washington, DC, USA, 2007. IEEE Computer Society.

[16] R. A. Bringmann. Compiler-Controlled Speculation. PhD thesis, University of Illinois, Urbana, IL, 1995.

[17] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. IEEE Transaction on Computers, C-35(8):677–691, August 1986.

[18] Martin Burtscher and Nana Sam. Automatic generation of high-performance trace compressors. In Proceedings of the 2005 International Conference on Code Generation and Optimization, pages 229–240, 2005.

[19] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The impact of synchronization and granularity on parallel systems. In Proceedings of the 17th annual international symposium on Computer Architecture, ISCA '90, pages 239–248, New York, NY, USA, 1990. ACM.

[20] Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew. Data dependence profiling for speculative optimizations. In Evelyn Duesterwald, editor, Compiler Construction, volume 2985 of Lecture Notes in Computer Science, pages 2733–2733. Springer Berlin / Heidelberg, 2004.

[21] J. Cogswell. Intel parallel studio 2011 helps out with the hard work. eWeek, 27(17):20–22, 2010.

[22] K. D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In Conference Record of the 16th ACM Symposium on the Principles of Programming Languages, pages 45–59, January 1989.

[23] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In Proceedings of the 22nd international conference on Software engineering, ICSE '00, pages 439–448, New York, NY, USA, 2000. ACM.

[24] B.A. Davey and H.A. Priestley. Introduction to Lattices and Order. Cambridge University Press, 2nd edition, 2002.

[25] Jialin Dou and Marcelo Cintra. A compiler cost model for speculative parallelization. ACM Trans. Archit. Code Optim., 4, June 2007.

[26] K. Ebcioglu. A compilation technique for software pipelining of loops with conditional jumps. In Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture, pages 69–79, December 1987.

[27] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding program comprehension by static and dynamic feature analysis. Software Maintenance, IEEE International Conference on, 0:602, 2001.

[28] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, pages 242–256, June 1994.

[29] C. J. Fidge. Partial orders for parallel debugging. In Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging, PADD '88, pages 183–194, New York, NY, USA, 1988. ACM.

[30] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. IEEE Trans. Softw. Eng., 17(8):751–761, 1991.

[31] S. Garcia, D. Jeon, C. Louie, and M.B. Taylor. Kremlin: Rethinking and rebooting gprof for the multicore age. PLDI, 2011.

[32] John Giacomoni, Tipp Moseley, and Manish Vachharajani. FastForward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 43–52, 2008.

[33] J. Gilchrist. Parallel data compression with bzip2. In Parallel and Distributed Computing and Systems, November 2004.

[34] Ian Glendinning and Bernhard Ömer. Parallelization of the qc-lib quantum computer simulator library. In Roman Wyrzykowski, Jack Dongarra, Marcin Paprzycki, and Jerzy Wasniewski, editors, Parallel Processing and Applied Mathematics, volume 3019 of Lecture Notes in Computer Science, pages 461–468. Springer Berlin / Heidelberg, 2004.

[35] Justin E. Gottschlich, Manish Vachharajani, and Siek G. Jeremy. An efficient software transactional memory using commit-time invalidation. In Proceedings of the International Symposium on Code Generation and Optimization (CGO), apr 2010.

[36] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ASE '05, pages 263–272, New York, NY, USA, 2005. ACM.

[37] Gary D. Hachtel and Fabio Somenzi. Logic Synthesis and Verification Algorithms. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[38] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Interprocedural parallelization analysis in suif. ACM Trans. Program. Lang. Syst., 27(4):662–731, 2005.

[39] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 253–262, New York, NY, USA, 2006. ACM.

[40] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming. Elsevier, Inc., 2008.

[41] J. P. Hoeflinger. Interprocedural Parallelization Using Memory Classification Analysis. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1998.

[42] Shin ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In DAC '93: Proceedings of the 30th international Design Automation Conference, pages 272–277, New York, NY, USA, 1993. ACM.

[43] Shin ichi Minato. Zero-suppressed bdds and their applications. International Journal on Software Tools for Technology Transfer, 3(2):156–70, May 20.

[44] Matthew Iyer, Chinmay Ashok, Joshua Stone, Neil Vachharajani, Daniel A. Connors, and Manish Vachharajani. Finding parallelism for future EPIC machines. In Proceedings of the 4th Workshop on Explicitly Parallel Instruction Computing Techniques (EPIC), March 2005.

[45] D. Jeon, S. Garcia, C. Louie, and M.B. Taylor. Parkour: Parallel speedup estimates for serial programs. HOTPAR, 2011.

[46] R. B. Jones and V. H. Allan. Software pipelining: An evaluation of Enhanced Pipelining. In Proceedings of the 24th International Workshop on Microprogramming and Microarchitecture, pages 82–92, November 1991.

[47] Mariam Kamkar, Peter Fritzson, and Nahid Shahmehri. Interprocedural dynamic slicing applied to interprocedural data how testing. In ICSM '93: Proceedings of the Conference on Software Maintenance, pages 386–395, Washington, DC, USA, 1993. IEEE Computer Society.

[48] Arun Kejariwal, Xinmin Tian, Milind Girkar, Wei Li, Sergey Kozhukhov, Utpal Banerjee, Alexander Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using spec cpu 2006. In Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '07, pages 215–225, New York, NY, USA, 2007. ACM.

[49] K. Kennedy, K.S. McKinley, and C.-W. Tseng. Interactive parallel programming using the parascope editor. Parallel and Distributed Systems, IEEE Transactions on, 2(3):329 –341, jul 1991.

[50] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. Sd3: A scalable approach to dynamic data-dependence profiling. Microarchitecture, IEEE/ACM International Symposium on, 0:535–546, 2010.

[51] X. Kong, D. Klappholz, and K. Psarris. The i test: An improved dependence test for automatic parallelization and vectorization. Parallel and Distributed Systems, IEEE Transactions on, 2(3):342–349, 1991.

[52] B. Korel and J. Laski. Dynamic program slicing. Inf. Process. Lett., 29(3):155–163, 1988.

[53] Matthias Krause. Bdd-based cryptanalysis of keystream generators. 2002.

[54] Jens Krinke. Slicing, chopping, and path conditions with barriers. Software Quality Control, 12:339–360, December 2004.

[55] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In Proceedings of the 19th International Symposium on Computer Architecture, pages 46–57, May 1992.

[56] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 1–12, New York, NY, USA, 2005. ACM.

[57] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 21:558–565, July 1978.

[58] James R. Larus. Whole program paths. In Proceedings of the SIGPLAN '99 Conference on Programming Languages Design and Implementation (PLDI 99), pages 259–269, May 1999.

[59] James R. Larus and Ravi Rajwar. Transactional Memory. Morgan and Claypool, 2006.

[60] J.R. Larus. Loop-level parallelism in numeric and symbolic programs. Parallel and Distributed Systems, IEEE Transactions on, 4(7):812–826, 1993.

[61] Doug Lea. Concurrent Programming in Java: Design Principles and Patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[62] Ondrej Lhoták, Stephen Curial, and Jose Nelson Amaral. Using zbdds in points-to analysis. 5234/2008:338–352, August 2008.

[63] Kathleen A. Lindlan, Janice Cuny, Allen D. Malony, Sameer Shende, Forschungszentrum Juelich, Reid Rivenburgh, Craig Rasmussen, and Bernd Mohr. A tool framework for static and dynamic analysis of object-oriented software with templates. In Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[64] P. Marcuello and A. Gonzalez. A quantitative assessment of thread-level speculation techniques. In Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International, pages 595 –601, 2000.

[65] Alan Mishchenko. An introduction to zero-suppressed binary decision diagrams. 2001.

[66] S. Muchnick. Advanced Compiler Design and Implementation. Morgan-Kaufmann Publishers, San Francisco, CA, 1997.

[67] Craig G. Nevill-Manning and Ian H. Witten. Linear-time, incremental hierarchy inference for compression. In DCC '97: Proceedings of the Conference on Data Compression, page 3, Washington, DC, USA, 1997. IEEE Computer Society.

[68] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[69] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[70] Chang Yun Park. Predicting program execution times by analyzing static and dynamic program paths. Real-Time Syst., 5:31–62, March 1993.

[71] Insung Park, Michael Voss, Seon Wook Kim, and Rudolf Eigenmann. Parallel programming environment for openMP. Scientific Programming, 9(2-3):143–161, 2001.

[72] M. Postiff, G. Tyson, and T. Mudge. Performance limits of trace caches. Technical Report CSE-TR-373-98, University of Maryland, Department of Electrical Engineering and Computer Science, CSE, September 1998.

[73] Manohar K. Prabhu and Kunle Olukotun. Using thread-level speculation to simplify manual parallelization. In Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '03, pages 1–12, New York, NY, USA, 2003. ACM.

[74] Manohar K. Prabhu and Kunle Olukotun. Exposing speculative thread parallelism in spec2000. In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '05, pages 142–152, New York, NY, USA, 2005. ACM.

[75] Graham Price and Manish Vachharajani. A case for compressing traces with BDDs. Computer Architecture Letters, 5, November 2006.

[76] Graham Price and Manish Vachharajani. Large program trace analysis and compression with zdds. In Proceedings of the International Symposium on Code Generation and Optimization (CGO), apr 2010.

[77] Graham D. Price. Enabling advanced program analysis with bdds. Master's thesis, Department of Electrical and Computer Engineering, University of Colorado at Boulder, Boulder, CO, 2008.

[78] Graham D. Price, John Giacomoni, and Manish Vachharajani. Visualizing potential parallelism in sequential programs. In PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, pages 82–90, New York, NY, USA, 2008. ACM.

[79] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. In Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems, ASPLOS '10, pages 65–76, New York, NY, USA, 2010. ACM.

[80] Ravi Ramaseshan and Frank Mueller. Toward thread-level speculation for coarse-grained parallelism with regular access patterns. 2008.

[81] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In 13th International Conference on Parallel Architectures and Compilation Techniques, pages 177–188, September 2004.

[82] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pages 177–188, Washington, DC, USA, 2004. IEEE Computer Society.

[83] Steven P. Reiss and Manos Renieris. Encoding program executions. In ICSE '01: Proceedings of the 23rd International Conference on Software Engineering, pages 221–230, Washington, DC, USA, 2001. IEEE Computer Society.

[84] F. Somenzi. CUDD: Colorado University Decision Diagram package, release 2.42. Technical report, University of Colorado at Boulder, http://vlsi.colorado.edu/~fabio/CUDD/, 2009.

[85] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, pages 112–122, New York, NY, USA, 2007. ACM Press.

[86] B. Steensgaard. Points-to analysis in almost linear time. In Proceedings of the ACM Symposium on Principles of Programming Languages, pages 32–41, January 1996.

[87] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In Proceedings of the 27th International Symposium on Computer Architecture, pages 1–12, June 2000.

[88] H. SUTTER. The free lunch is over : A fundamental turn toward concurrency in software. Dr. Dobb's Journal, 2005.

[89] Alred Tarski. A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics, 5(2):285–309, 1955.

[90] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[91] Frank Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[92] R. A. Towle. Control and Data Dependence for Program Transformations. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, 1976.

[93] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. Parallel Architectures and Compilation Techniques, International Conference on, 0:49–59, 2007.

[94] VectorFabrics. vfanalyst: Analyze your sequential c code to create an optimized parallel implementation, May 2011.

[95] Blu W. and R. Eigenmann. Performance analysis of parallelizing compilers on the perfect benchmarks programs. IEEE Trans. Parallel Distrib. Syst., 3:643–656, November 1992.

[96] David W. Wall. Limits of instruction-level parallelism. Technical Report 93/6, DEC WRL, November 1993.

[97] 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 23rd international conference on Supercomputing, ICS '09, pages 158–168, New York, NY, USA, 2009. ACM.

[98] Fredrik Warg and Per Stenstr. Limits on speculative module-level parallelism in imperative and object-oriented programs on cmp platforms. volume 0, page 0221. Los Alamitos, CA, USA, 2001.

[99] John Whaley. Context-sensitive pointer analysis using binary decision diagrams. PhD thesis, Stanford, CA, USA, 2007. Adviser-Lam, Monica.

[100] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, pages 131–144, New York, NY, USA, 2004. ACM Press.

[101] Peng Wu, Arun Kejariwal, and Călin Caşcaval. Languages and compilers for parallel computing. chapter Compiler-Driven Dependence Profiling to Guide Program Parallelization, pages 232–248. Springer-Verlag, Berlin, Heidelberg, 2008.

[102] Y. Wu and J. R. Larus. Static branch prediction and program profile analysis. In Proceedings of the 27th Annual International Symposium on Microarchitecture, pages 1–11, December 1994.

[103] Xiangyu Zhang and Rajesh Gupta. Whole execution traces. In 37th International Symposium on Microarchitecture (MICRO-37), pages 105–116, 2004.

[104] Xiangyu Zhang, Rajesh Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In Proceedings of the 25th International Conference on Software Engineering (ICSE), pages 319–329, May 2003.

[105] Xiangyu Zhang, Rajesh Gupta, and Youtao Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In 26th International conference on Software Engineering (ICSE-26), pages 502–511, 2004.

[106] Xiangyu Zhang, Armand Navabi, and Suresh Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '09, pages 47–58, Washington, DC, USA, 2009. IEEE Computer Society.

Appendix A

ZDD Trace Data Types

The DIN_i is a data member of the $\{DIN_i, DIN_d\}$ tuple set. It represents an individual dynamic instructions. The DIN_d data member is the set of dynamic instructions that have a true data dependency resolved by DIN_i. The members of DIN_d must be scheduled to execute after DIN_i has executed.

The SIN data type represents a static instruction number. It also is the instruction address in the original binary that was used for trace creation. Therefore, it is possible to find the original source code line using a SIN value.

The RDY time is the earliest time at which a dynamic instruction may be scheduled to execute on an optimal machine. An optimal machine has perfect branch prediction, an unlimited instruction window, an infinite number of functional units, single cycle execution of all instructions, and more.

A HOT data member is the number of times a static instruction (or SIN) was executed during trace collection. It is equivalent to the number of DIN values that exist for a given SIN.

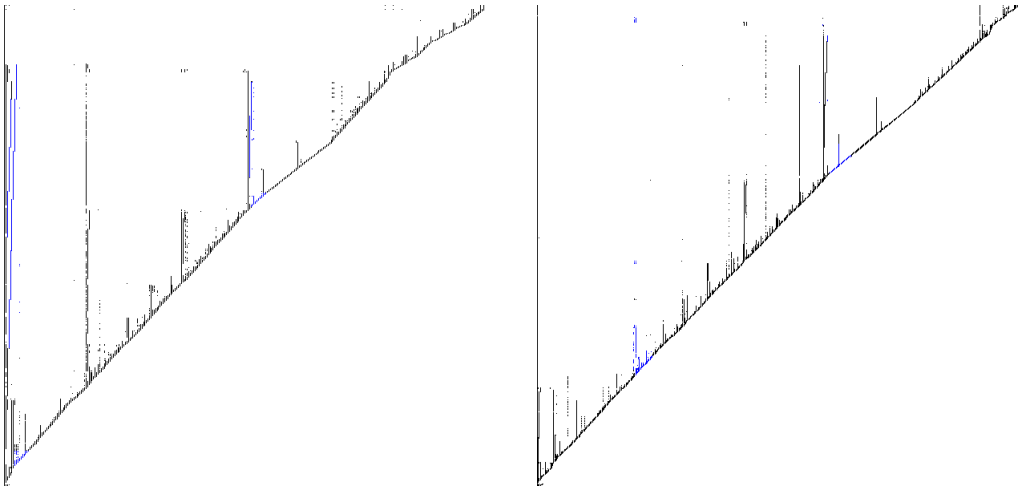
The SYS data type contains system call values on a Linux version 2.6.24-28-server using the 64-bit set of system calls. Figure A.1 contains all tuple types used in this thesis, and the relation type.

Tuple	Set Type
$\{DIN_i, DIN_d\}$	Surjective $DIN_d \Rightarrow DIN_i$
$\{DIN, SIN\}$	Surjective $DIN \Rightarrow SIN$
$\{DIN, RDY\}$	Bijective
$\{DIN, HOT\}$	Surjective $DIN \Rightarrow HOT$
$\{DIN, SYS\}$	Surjective $DIN \Rightarrow SYS$

Figure A.1: Tuple Relation Types

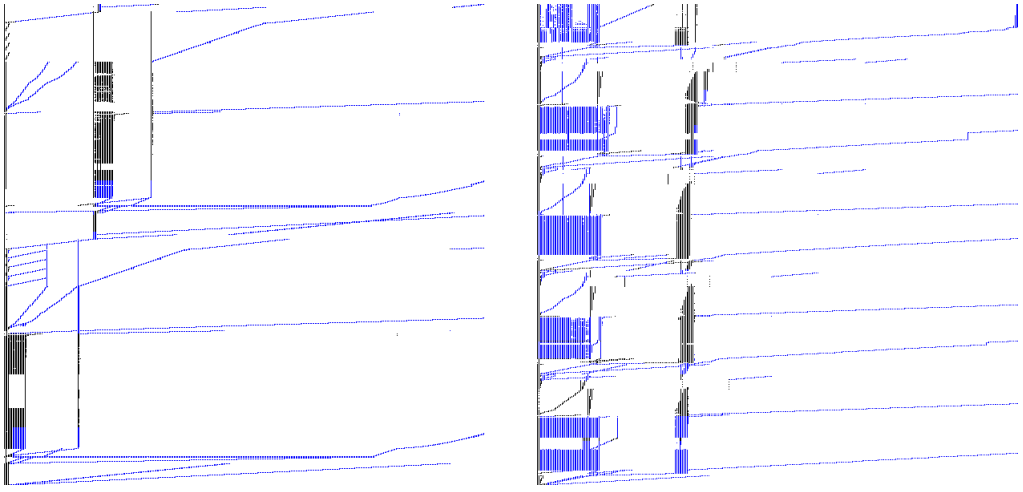
Appendix B

Selected Regions from SPEC 2006



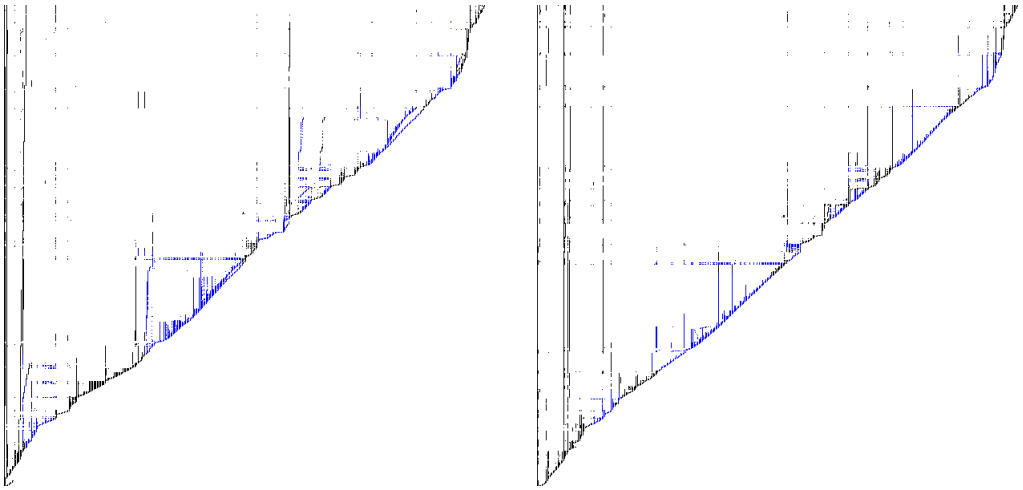
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure B.1: 400.perlbench with Selected Regions



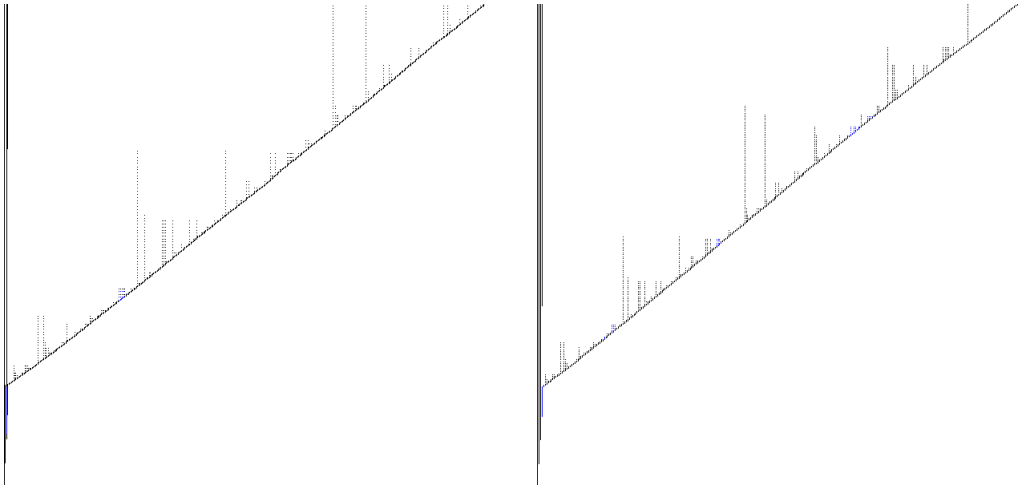
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure B.2: 401.bzip2 with Selected Regions



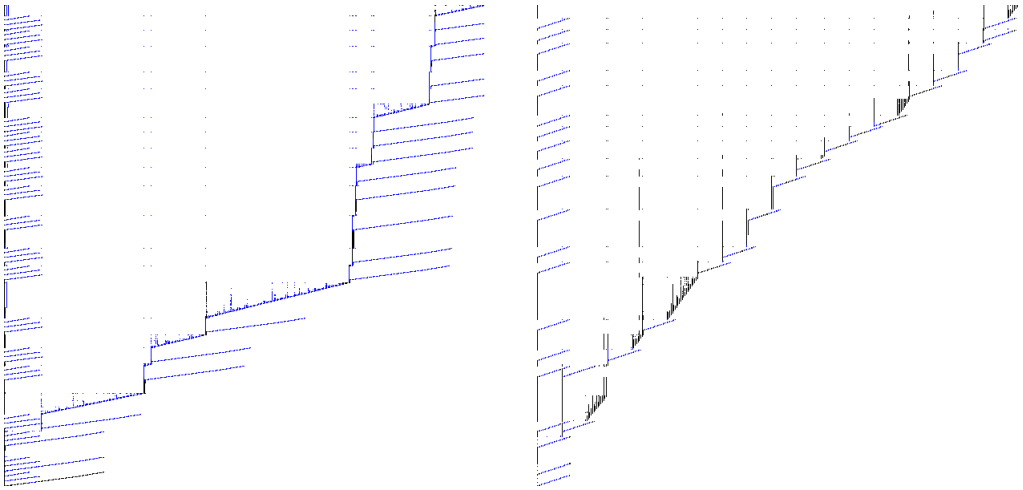
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure B.3: 403.gcc with Selected Regions



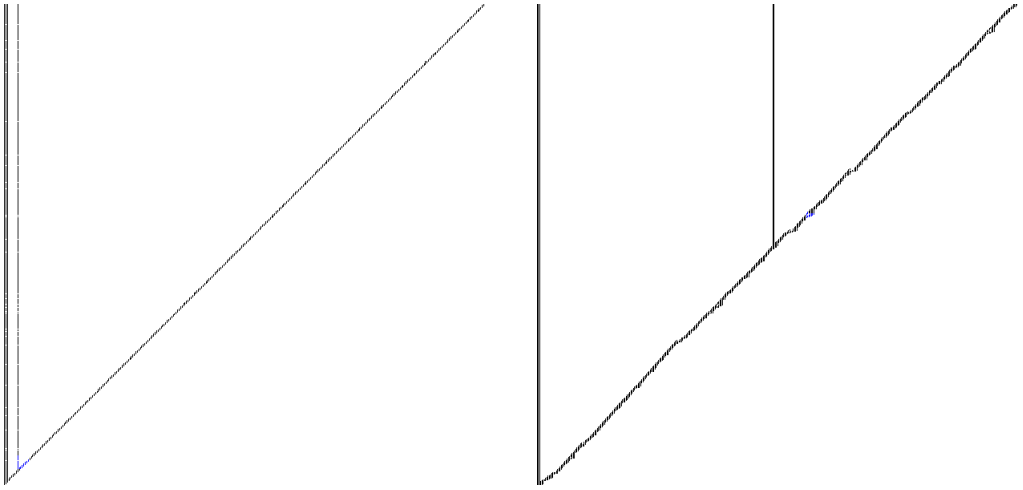
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure B.4: 429.mcf with Selected Regions



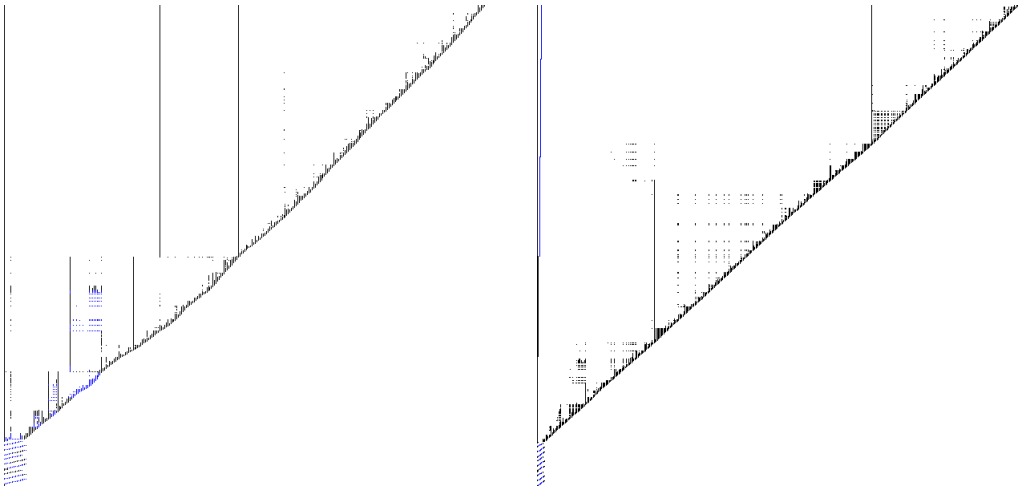
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure B.5: 445.go with Selected Regions



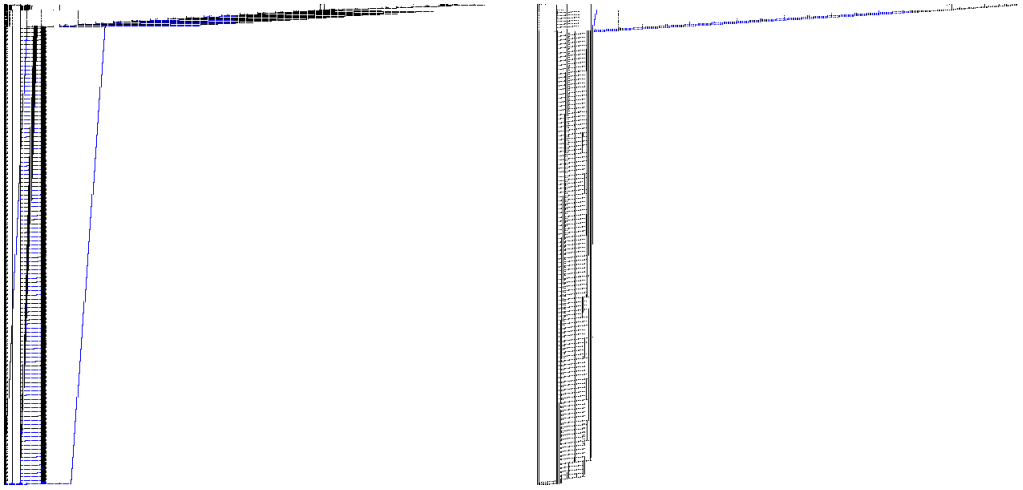
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure B.6: 456.hmmer with Selected Regions



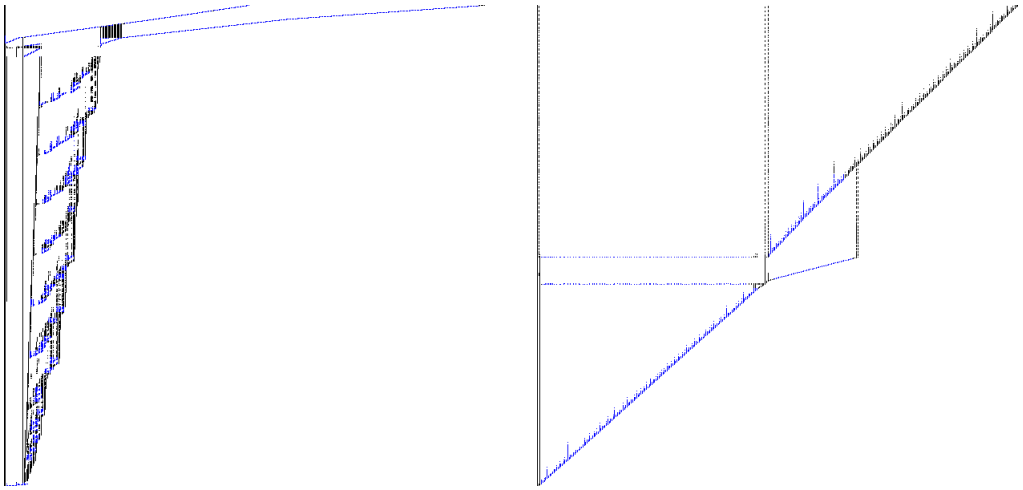
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure B.7: 458.sjeng with Selected Regions



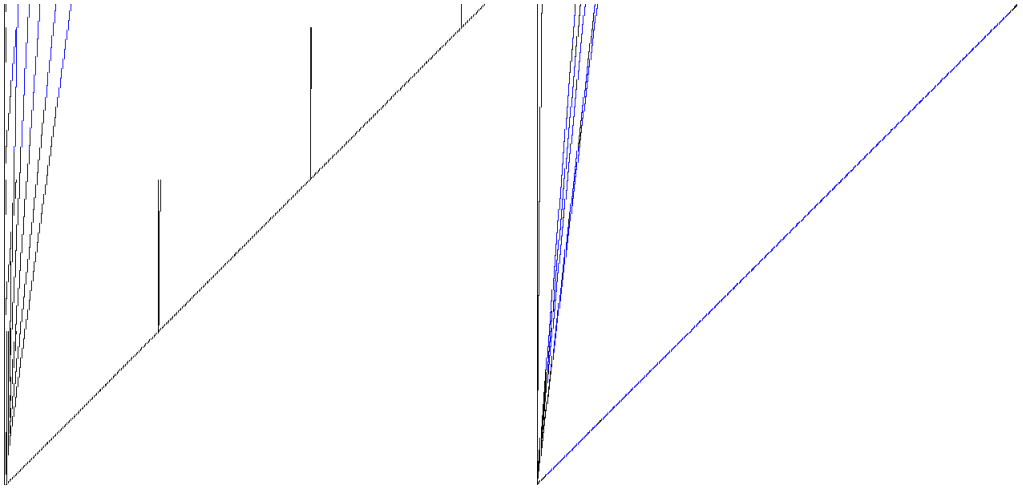
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure B.8: 462.libquantum with Selected Regions



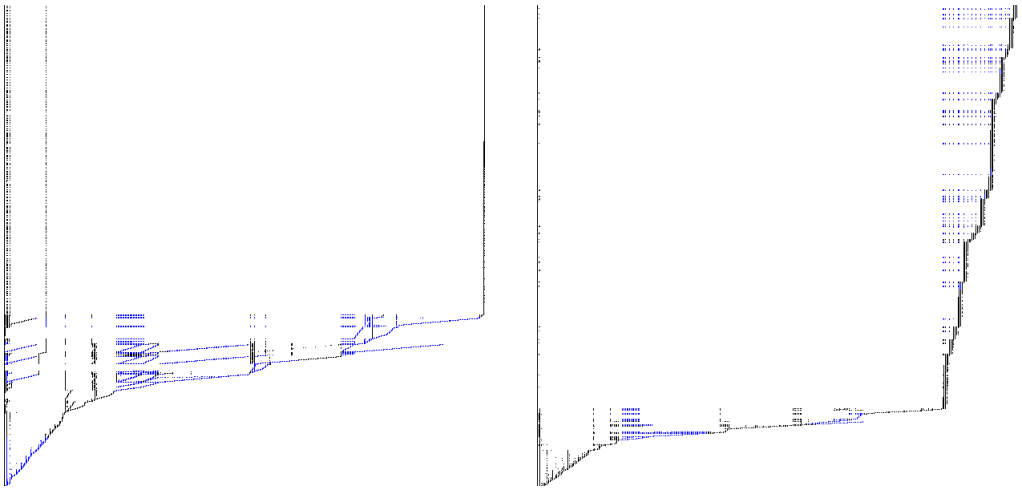
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure B.9: 464.h264ref with Selected Regions



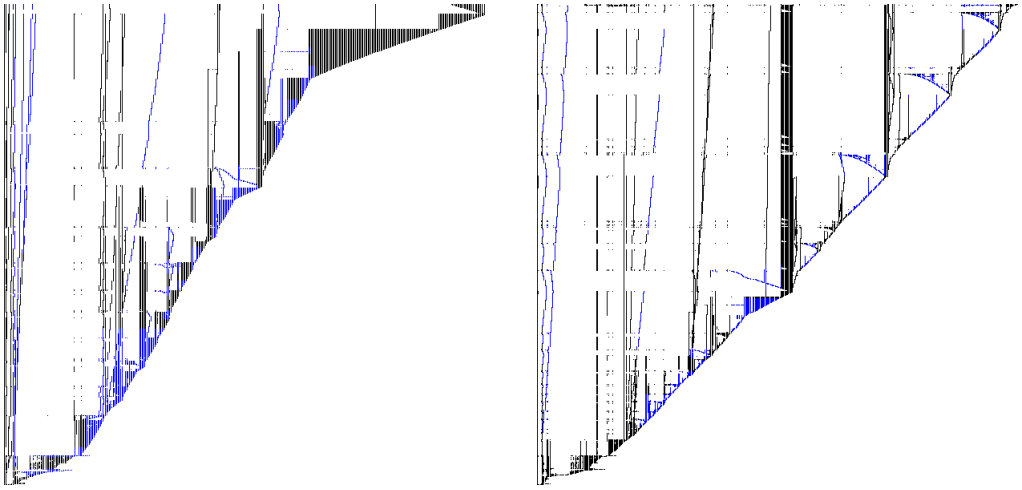
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure B.10: 471.omnetpp with Selected Regions



(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure B.11: 473.astar with Selected Regions



(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure B.12: 483.xalancbmk with Selected Regions

Appendix C

Hot Code Visualizations of SPEC 2000

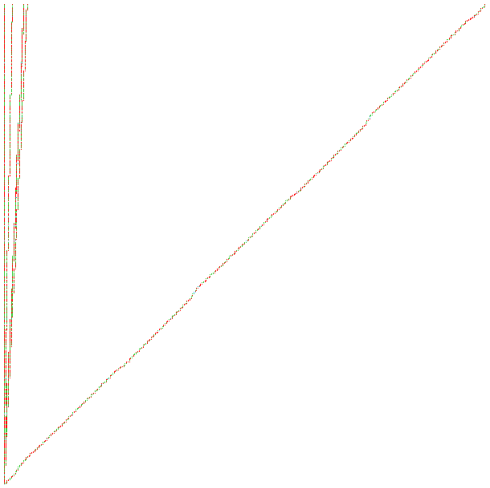


Figure C.1: 164.gzip DINxRDY with Hot Code

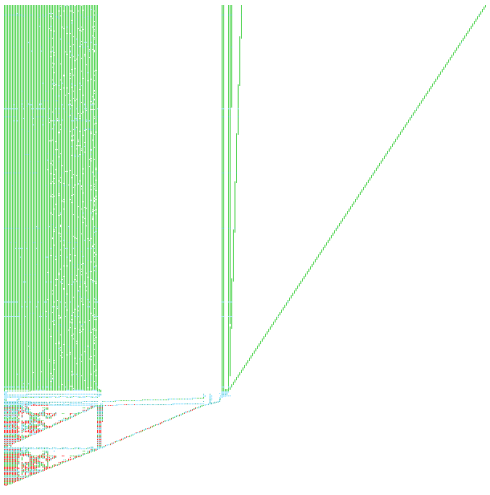


Figure C.2: 175.vpr DINxRDY with Hot Code

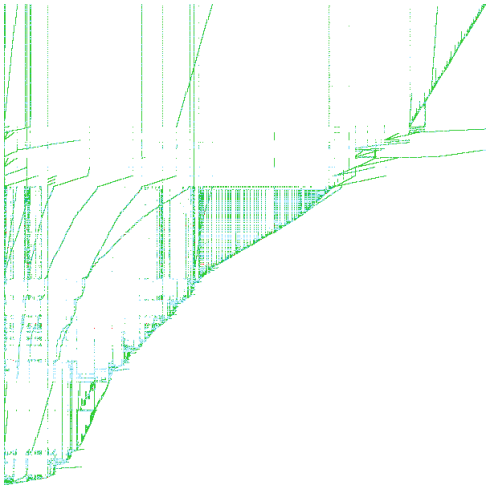


Figure C.3: 176.gcc DINxRDY with Hot Code

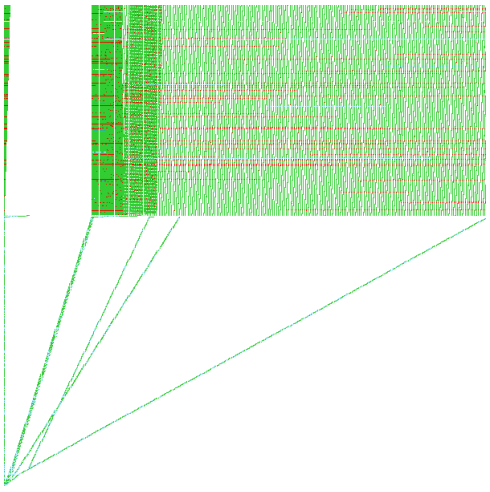


Figure C.4: 181.mcf DINxRDY with Hot Code

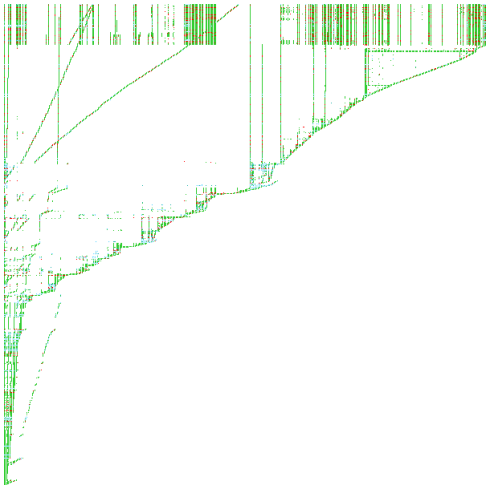


Figure C.5: 197.parser DINxRDY with Hot Code

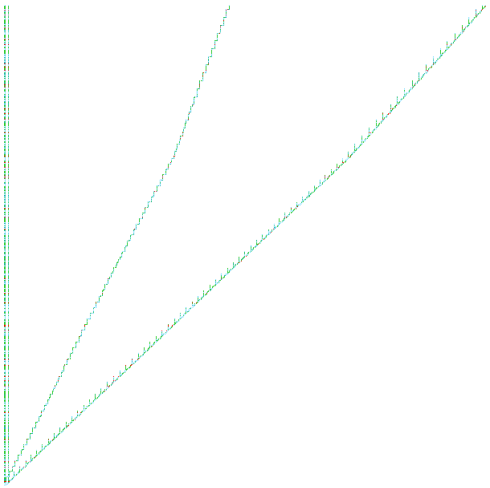


Figure C.6: 252.eon DINxRDY with Hot Code

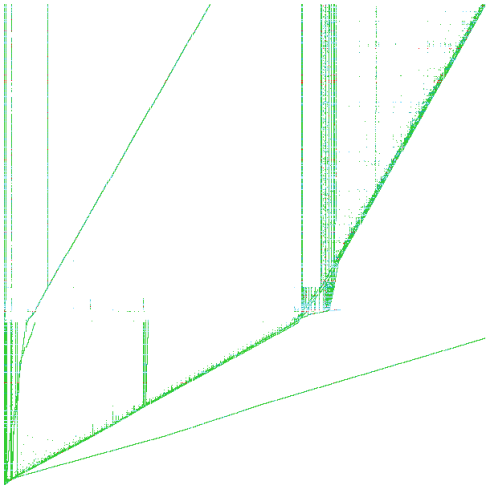


Figure C.7: 253.perl DINxRDY with Hot Code

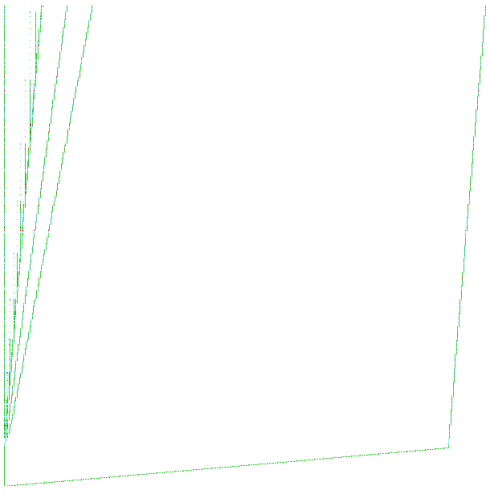


Figure C.8: 254.gap DINxRDY with Hot Code

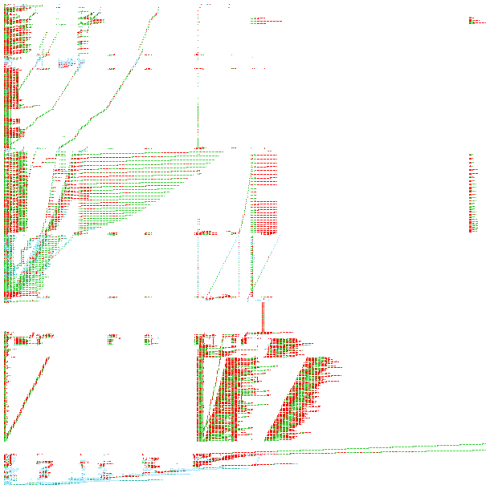


Figure C.9: 255.vortex DINxRDY with Hot Code

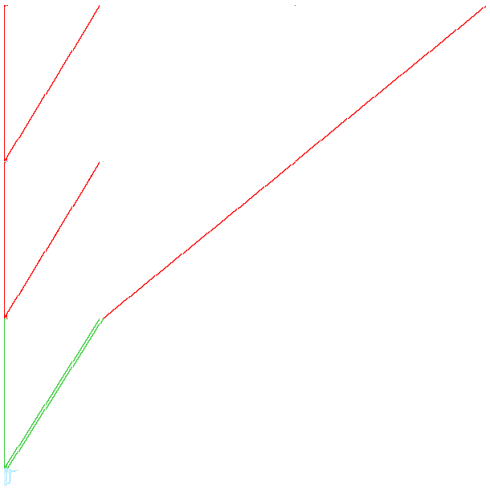


Figure C.10: 256bzip2 DINxRDY with Hot Code

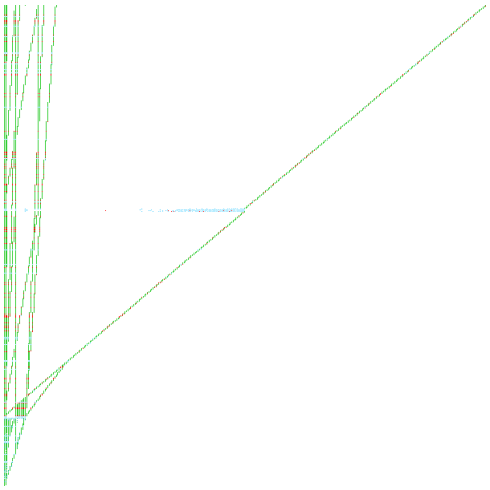
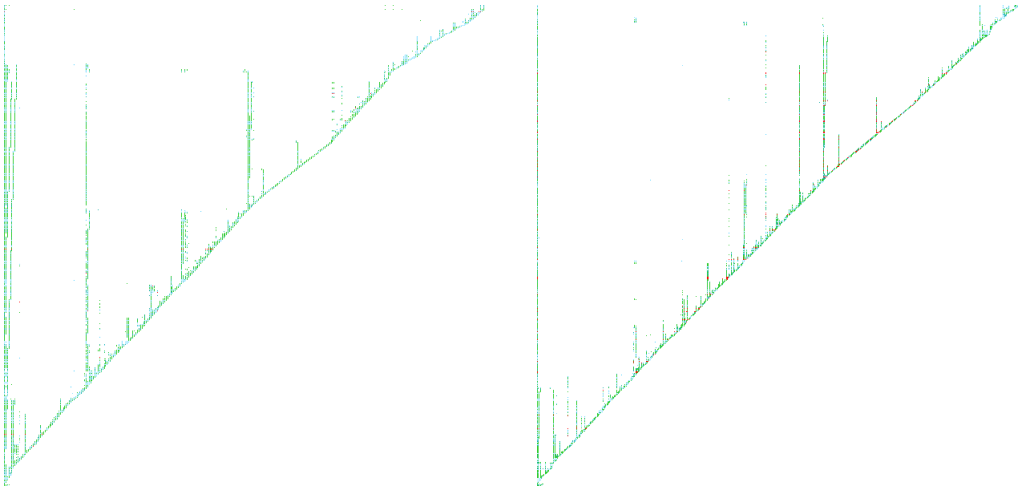


Figure C.11: 300.twolf DINxRDY with Hot Code

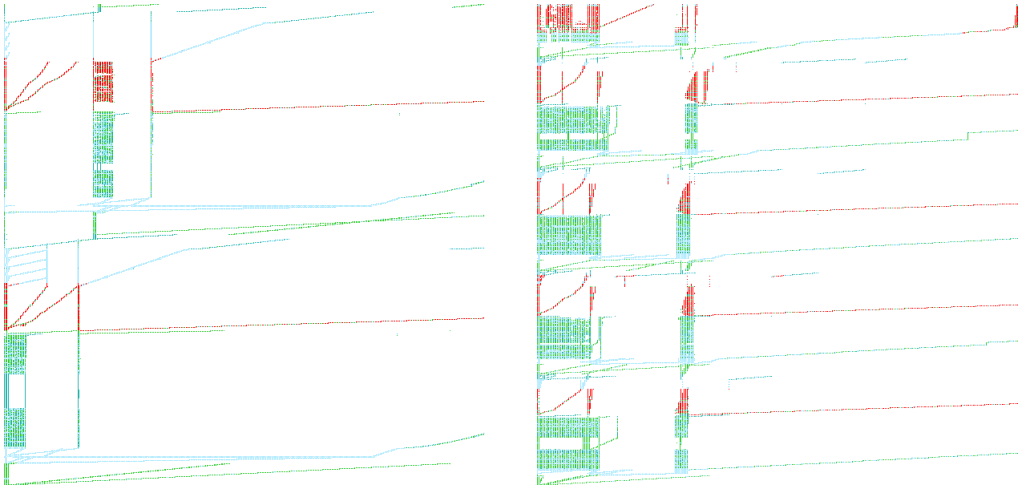
Appendix D

Hot Code Visualizations of SPEC 2006



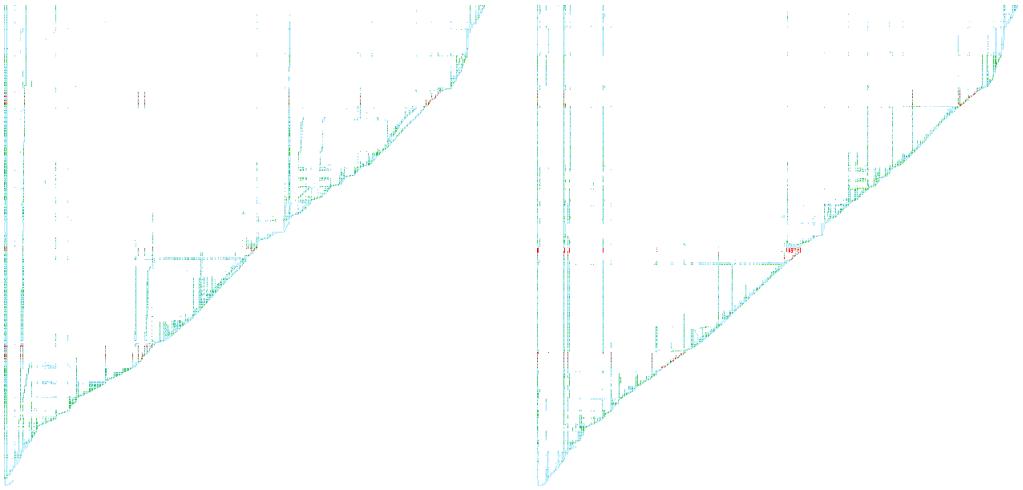
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure D.1: 400.perlbench DINxRDY with Hot Code



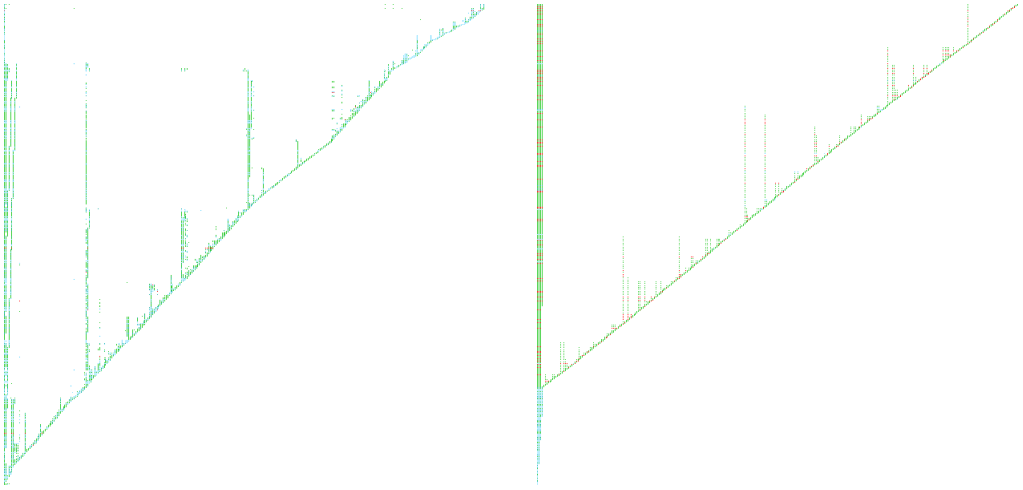
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure D.2: 401.bzip2 DINxRDY with Hot Code



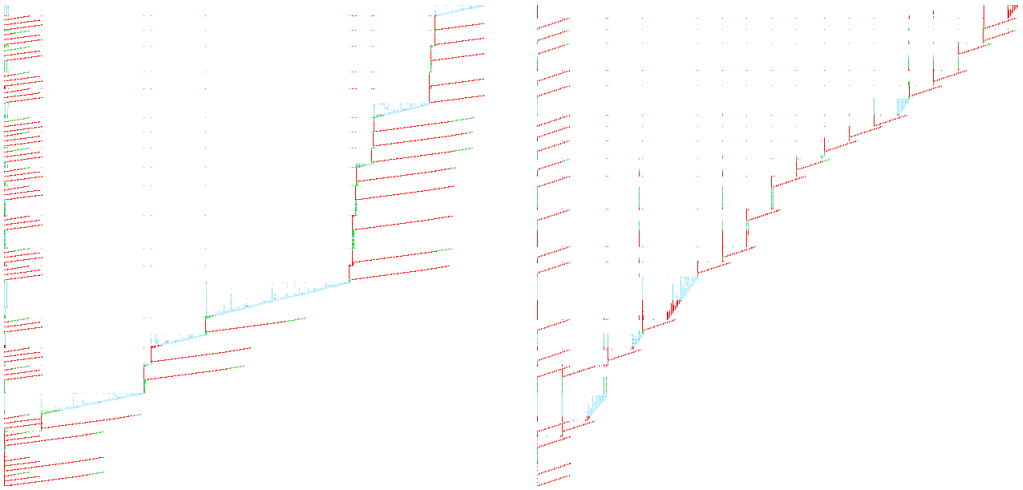
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure D.3: 403.gcc DINxRDY with Hot Code



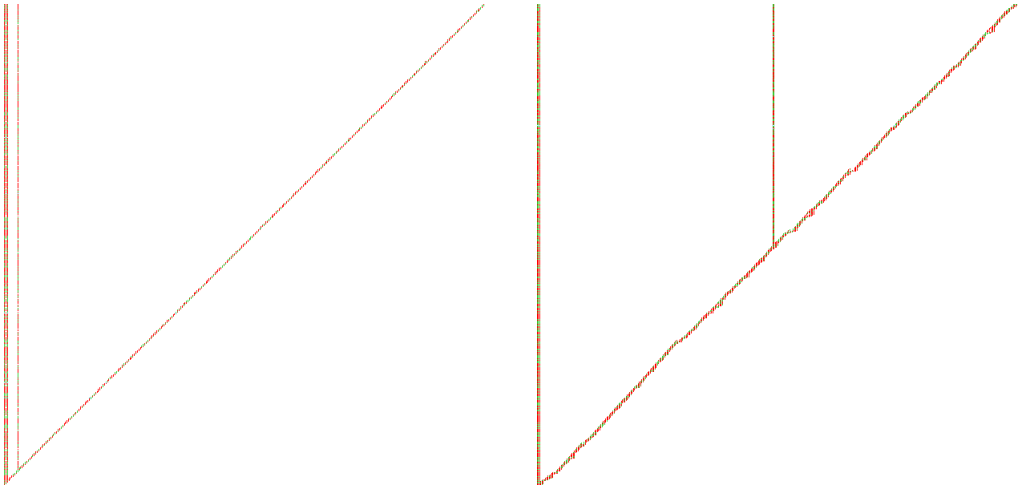
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure D.4: 429.mcf DINxRDY with Hot Code



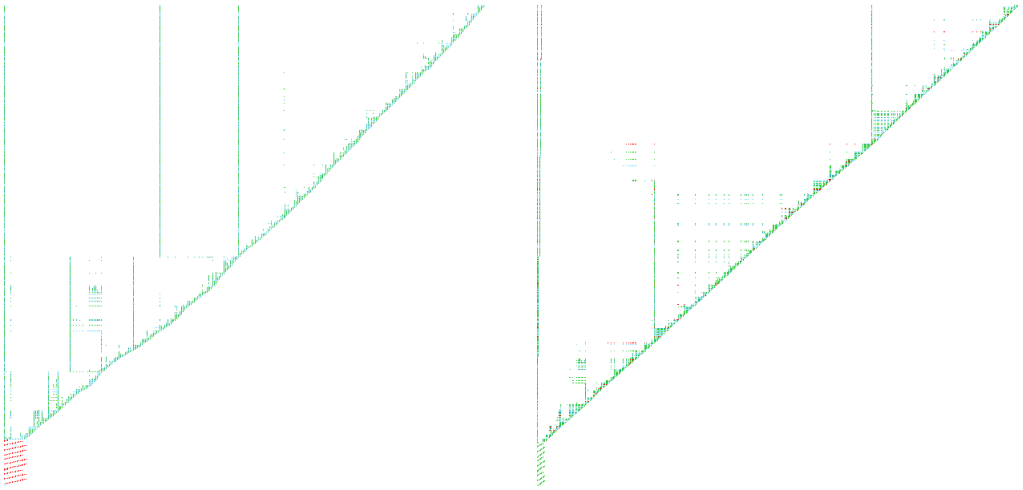
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure D.5: 445.go DINxRDY with Hot Code



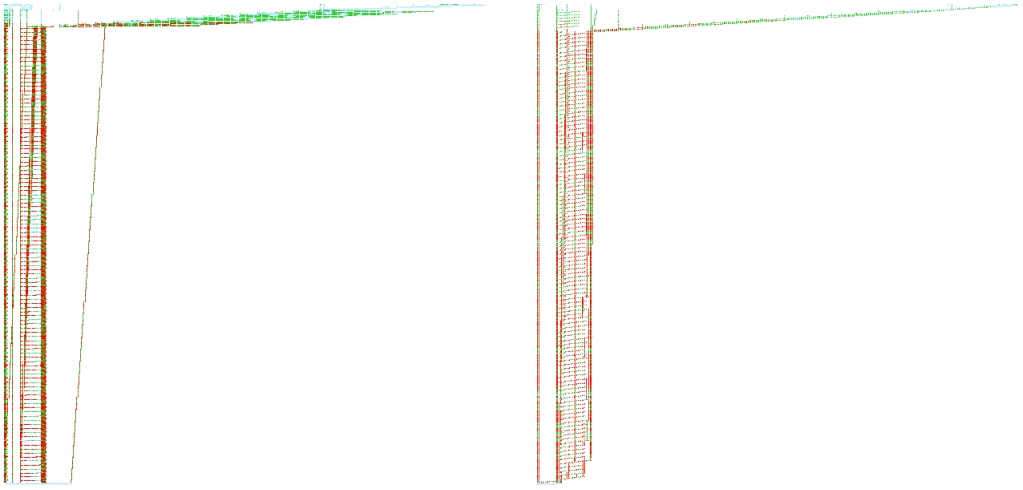
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure D.6: 456.hmmer DINxRDY with Hot Code



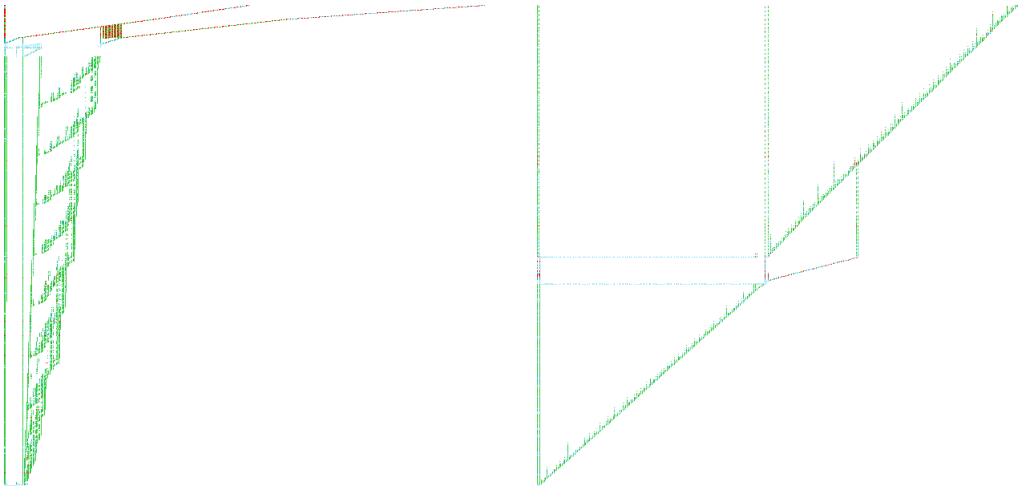
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure D.7: 458.sjeng DINxRDY with Hot Code



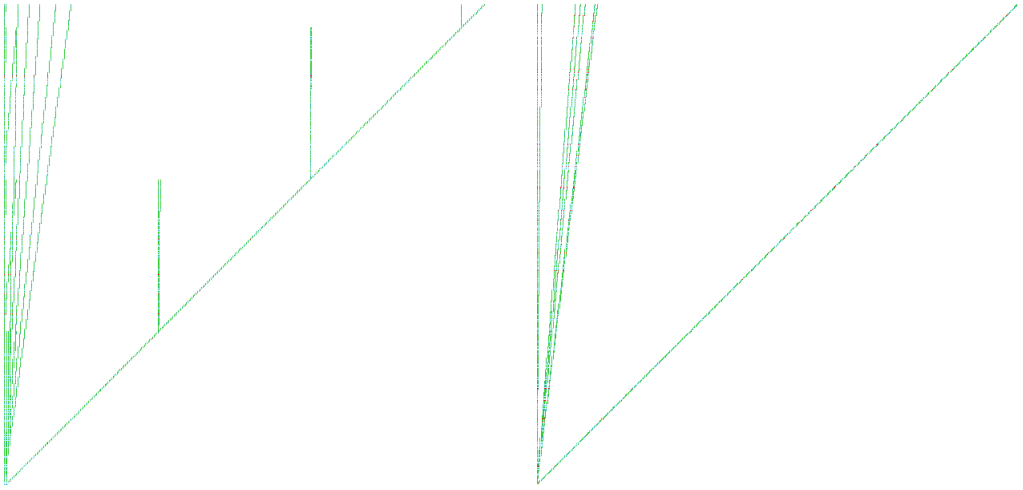
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure D.8: 462.libquantum DINxRDY with Hot Code



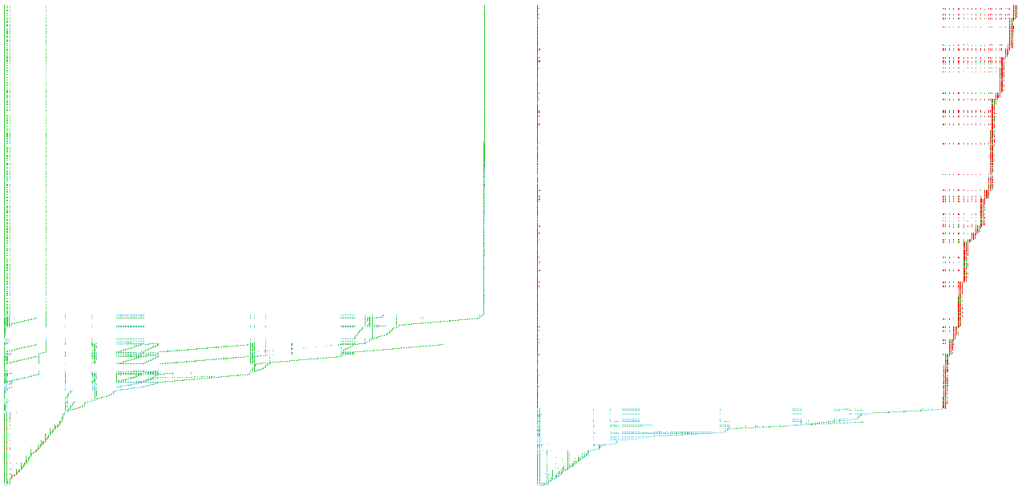
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure D.9: 464.h264ref DINxRDY with Hot Code



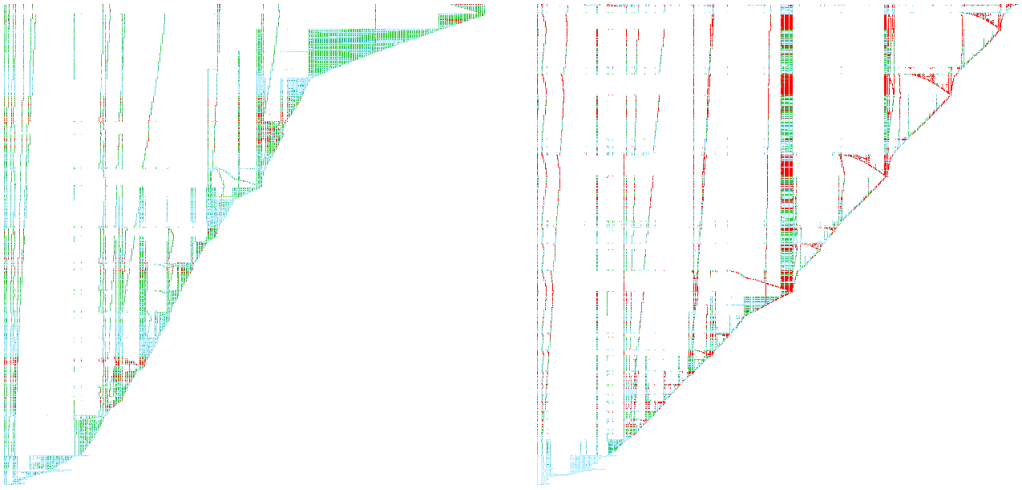
(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure D.10: 471.omnetpp DINxRDY with Hot Code



(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure D.11: 473.astar DINxRDY with Hot Code



(a) GCC -O0 DINxRDY (b) GCC -O2 DINxRDY

Figure D.12: 483.xalancbmk DINxRDY with Hot Code

Appendix E

Trace ZDD and BDD Variable Orders

Variable Index	Naïve LSB-MSB	Naïve MSB-LSB	LSB-MSB	MSB-LSB	SPLIT
0	0	127	0	127	93
1	1	126	64	63	29
2	2	125	1	126	92
3	3	124	65	62	28
4	4	123	2	125	91
5	5	122	66	61	27
6	6	121	3	124	90
7	7	120	67	60	26
8	8	119	4	123	89
9	9	118	68	59	25
10	10	117	5	122	88
11	11	116	69	58	24
12	12	115	6	121	87
13	13	114	70	57	23
14	14	113	7	120	86
15	15	112	71	56	22
16	16	111	8	119	85
17	17	110	72	55	21
18	18	109	9	118	84
19	19	108	73	54	20
20	20	107	10	117	83
21	21	106	74	53	19

22	22	105	11	116	82
23	23	104	75	52	18
24	24	103	12	115	81
25	25	102	76	51	17
26	26	101	13	114	80
27	27	100	77	50	16
28	28	99	14	113	79
29	29	98	78	49	15
30	30	97	15	112	78
31	31	96	79	48	14
32	32	95	16	111	77
33	33	94	80	47	13
34	34	93	17	110	76
35	35	92	81	46	12
36	36	91	18	109	75
37	37	90	82	45	11
38	38	89	19	108	74
39	39	88	83	44	10
40	40	87	20	107	73
41	41	86	84	43	9
42	42	85	21	106	72
43	43	84	85	42	8
44	44	83	22	105	71
45	45	82	86	41	7
46	46	81	23	104	70
47	47	80	87	40	6
48	48	79	24	103	69
49	49	78	88	39	5
50	50	77	25	102	68
51	51	76	89	38	4
52	52	75	26	101	67

53	53	74	90	37	3
54	54	73	27	100	66
55	55	72	91	36	2
56	56	71	28	99	65
57	57	70	92	35	1
58	58	69	29	98	64
59	59	68	93	34	0
60	60	67	30	97	127
61	61	66	94	33	63
62	62	65	31	96	126
63	63	64	95	32	62
64	64	63	32	95	125
65	65	62	96	31	61
66	66	61	33	94	124
67	67	60	97	30	60
68	68	59	34	93	123
69	69	58	98	29	59
70	70	57	35	92	122
71	71	56	99	28	58
72	72	55	36	91	121
73	73	54	100	27	57
74	74	53	37	90	120
75	75	52	101	26	56
76	76	51	38	89	119
77	77	50	102	25	55
78	78	49	39	88	118
79	79	48	103	24	54
80	80	47	40	87	117
81	81	46	104	23	53
82	82	45	41	86	116
83	83	44	105	22	52

84	84	43	42	85	115
85	85	42	106	21	51
86	86	41	43	84	114
87	87	40	107	20	50
88	88	39	44	83	113
89	89	38	108	19	49
90	90	37	45	82	112
91	91	36	109	18	48
92	92	35	46	81	111
93	93	34	110	17	47
94	94	33	47	80	110
95	95	32	111	16	46
96	96	31	48	79	109
97	97	30	112	15	45
98	98	29	49	78	108
99	99	28	113	14	44
100	100	27	50	77	107
101	101	26	114	13	43
102	102	25	51	76	106
103	103	24	115	12	42
104	104	23	52	75	105
105	105	22	116	11	41
106	106	21	53	74	104
107	107	20	117	10	40
108	108	19	54	73	103
109	109	18	118	9	39
110	110	17	55	72	102
111	111	16	119	8	38
112	112	15	56	71	101
113	113	14	120	7	37
114	114	13	57	70	100

115	115	12	121	6	36
116	116	11	58	69	99
117	117	10	122	5	35
118	118	9	59	68	98
119	119	8	123	4	34
120	120	7	60	67	97
121	121	6	124	3	33
122	122	5	61	66	96
123	123	4	125	2	32
124	124	3	62	65	95
125	125	2	126	1	31
126	126	1	63	64	94
127	127	0	127	0	30

Appendix F

Functions from Parallel Region Selection

F.1 400.perlbmk -O0 Functions in Selected Regions for TLP

```
S_new_xpvbm
```

F.2 400.perlbmk -O2 Functions in Selected Regions for TLP

```
stat
PerlIOBase_p...
PerlIOBase_cl...
PerlIOBase_no...
PerlIOBase_re...
PerlIOBase_fi...
PerlIOBuf_clo...
PerlIOBuf_get...
PerlIOBuf_fi...
PerlIOBuf_fl...
PerlIOBuf_ope...
PerlIOBuf_pu...
PerlIOBuf_po...
PerlIOBuf_set...
PerlIOBuf_rea...
PerlIOBuf_tel...
PerlIOUnix_clo...
```

```
PerlIOUnix_fil...
PerlIOUnix_op...
PerlIOUnix_of...
PerlIOUnix_pus...
PerlIOUnix_se...
PerlIOUnix_re...
PerlIOUnix_tel...
PerlIO_arg_fet...
PerlIO_alloca...
PerlIO_context...
PerlIO_debug
PerlIO_defaul...
PerlIO_getc
PerlIO_fast_get...
PerlIO_isutf8
PerlIO_layer_f...
PerlIO_list_fr...
PerlIO_openn
PerlIO_push
PerlIO_pop
PerlIO_resolve...
PerlIO__close
Perl_append_ele...
Perl_append_lis...
Perl_allocmy
Perl_av_clear
Perl_av_extend
Perl_av_fetch
Perl_av_make
Perl_av_push
```

```
Perl_av_shift
Perl_av_store
Perl_av_undef
Perl_cast_ul...
Perl_call_sv
Perl_call_li...
Perl_ck_conca...
Perl_ck_bitop
Perl_ck_exist...
Perl_ck_fun
Perl_ck_join
Perl_ck_metho...
Perl_ck_null
Perl_ck_s...
Perl_ck_requ...
Perl_ck_rvco...
Perl_croak
Perl_convert
Perl_cv_undef
Perl_block_start
Perl_block_end
Perl_block_gimme
Perl_die_where
Perl_do_join
Perl_gp_ref
Perl_grok_numb...
Perl_grok_nume...
Perl_gv_AVadd
Perl_gv_HVadd
Perl_gv_stashpv...
```

```
Perl_gv_init
Perl_gv_fetch...
Perl_free_tmps
Perl_fbm_instr
Perl_force_lis...
Perl_fold_cons...
Perl_intro_my
Perl_hv_iterinit
Perl_hv_clear
Perl_hv_exists_ent
Perl_hv_store_ent
Perl_hv_fetch_ent
Perl_keyword
Perl_jmaybe
Perl_magic_set
Perl_magic_get
Perl_my_attrs
Perl_my_atof2
Perl_mod
Perl_mg_size
Perl_mg_set
Perl_mg_magica...
Perl_mg_get
Perl_mg_find
Perl_listkids
Perl_linklist
Perl_leave_sco...
Perl_lex_start
Perl_lex_end
Perl_localize
```

```
Perl_op_clear
Perl_op_null
Perl_op_free
Perl_oopsHV
Perl_new...
Perl_packa...
Perl_pad_a...
Perl_pad_ch...
Perl_pad_bl...
Perl_pad_f...
Perl_pad_le...
Perl_pad_ne...
Perl_pad_re...
Perl_pad_un...
Perl_pad_ti...
Perl_peep
Perl_pop_sco...
Perl_pop_ret...
Perl_pp_a...
Perl_pp_cal...
Perl_pp_con...
Perl_pp_bit_o...
Perl_pp_eq
Perl_pp_ex...
Perl_pp_en...
Perl_pp_defi...
Perl_pp_gr...
Perl_pp_gt
Perl_pp_gv...
Perl_pp_iter
```

```
Perl_pp_hsl...

Perl_pp_hel...

Perl_pp_join

Perl_pp_mat...

Perl_pp_met...

Perl_pp_li...

Perl_pp_le...

Perl_pp_lt

Perl_pp_oras...

Perl_pp_nex...

Perl_pp_not

Perl_pp_p...

Perl_pp_s...

Perl_pp_re...

Perl_pp_rv2...

Perl_pp_uc

Perl_pp_un...

Perl_prepe...

Perl_pregc...

Perl_push_sc...

Perl_push_re...

Perl_save...

Perl_sawpar...

Perl_safesy...

Perl_scalar...

Perl_scan_nu...

Perl_scope

Perl_start_sub...

Perl_sv_add_are...

Perl_sv_catpv...
```

```
Perl_sv_clear
Perl_sv_eq
Perl_sv_grow
Perl_sv_gets
Perl_sv_free
Perl_sv_force...
Perl_sv_magic...
Perl_sv_morta...
Perl_sv_newmor...
Perl_sv_set...
Perl_sv_2uv
Perl_sv_2cv
Perl_sv_2mor...
Perl_sv_upgrad...
Perl_sv_tainte...
Perl_sv_vcatp...
Perl_sv_vsetp...
Perl_sv_replac...
Perl_runops_stan...
Perl_regexec_f...
Perl_re_intuit...
Perl_ref
Perl_utilize
Perl_vcroak
Perl_vmess
Perl_vnewSVpvf
Perl_yyparse
Perl_yylex
Perl_PerlIO_close
Perl_PerlIO_get_pt...
```

```
Perl_PerlIO_get_cn...
Perl_PerlIO_get_ba...
Perl_PerlIO_fil...
Perl_PerlIO_flus...
Perl_PerlIO_see...
Perl_PerlIO_set...
Perl_PerlIO_stdi...
Perl_PerlIO_read
Perl_PerlIO_tell
S_ao
S_call_body
S_call_list_body
S_cop_free
S_expect_number
S_del_xiv
S_del_xpvav
S_del_xpviv
S_del_xpvcv
S_del_xpvmg
S_del_xpvnv
S_dopoptosub_a...
S_dopoptoloop
S_doeval
S_doopen_pm
S_gv_init_sv
S_filter_gets
S_force_ident
S_force_version
S_force_word
S_force_next
```

```
S_incline
S_intuit_method
S_intuit_more
S_hsplit
S_hv_fetch_common
S_hfreeentries
S_my_kid
S_mulexp10
S_method_common
S_more_he
S_more_sv
S_more_xpvav
S_more_xpvhv
S_more_xnv
S_modkids
S_list_assignment
S_lop
S_new_he
S_new_logop
S_new_xiv
S_new_xpvav
S_new_xpvcv
S_new_xpviv
S_new_xpvhv
S_new_xpvmg
S_new_xpvnv
S_new_xnv
S_pad_findlex
S_path_is_absolut...
S_pending_ident
```

```
S_save_hek_flag...
S_save_magic
S_share_hek_fla...
S_scalarboo...
S_scalar_mod...
S_scan_ident
S_scan_const
S_scan_str
S_scan_word
S_scan_formli...
S_sublex_push
S_sublex_start
S_sublex_done
S_skipspace
S_refto
S_tokeq
S_vdie_croak_common
restore_rsfp
restore_magic
yydestruct
XS_attributes_reft...
XS_attributes__modi...
XS_UNIVERSAL_can
```

F.3 401.bzip2 -O0 Functions in Selected Regions for TLP

```
add_pair_to_block
copy_input_until_s...
copy_output_until...
compressStream
BZ2_hbAssignCode...
```

```
BZ2_hbMakeCodeLe...
BZ2_compressBlock
BZ2_bzCompress
BZ2_bzWrite
BZ2_blockSort
generateMTFValue...
handle_compress
mainQSort3
mainSimple...
mainSort
mainGtU
makeMaps_e
myfeof
mmed3
prepare_new_block
spec_ungetc
spec_getc
spec_fread
spec_fwrite
sendMTFValues
bsPutUInt32
bsW
```

F.4 401.bzip2 -O2 Functions in Selected Regions for TLP

```
add_pair_to_block
copy_input_until_s...
copy_output_until...
compressStream
BZ2_hbAssignCode...
BZ2_hbMakeCodeLe...
```

```
BZ2_compressBlock
BZ2_bzCompress
BZ2_bzWrite
BZ2_blockSort
generateMTFValue...
handle_compress
mainQSort3
mainSimple...
mainSort
mainGtU
makeMaps_e
myfeof
mmed3
prepare_new_block
spec_ungetc
spec_getc
spec_fread
spec_fwrite
sendMTFValues
bsPutUInt32
bsW
```

F.5 403.gcc -O0 Functions in Selected Regions for TLP

```
active_insn...
adjust_off...
address...
addr_aff...
addr_sid...
add_insn_a...
add_insn_b...
```

aliases_e...

alias_set...

allocate...

alloc_INS...

alloc_pag...

alloc_blo...

alloc_EXP...

any_condjum...

any_uncondj...

approx_reg...

apply_chan...

apply_dis...

apply_del...

asm_noperan...

avoid_const...

cached_mak...

calculat...

calc_idom...

calc_dfs_t...

cancel...

canon_h...

canon_r...

can_com...

can_thr...

can_del...

can_fal...

cant_co...

change_addr...

check_asm_op...

check_hard_r...

```
check_depen...
check_promo...
check_for_la...
classify_i...
clear_edg...
cleanup_c...
close_dump...
copy_insn...
copy_rtx_i...
copy_loop...
comp...
combin...
commut...
count_reg...
count_bas...
const...
contro...
conver...
cselib_subs...
cselib_look...
cse_around_l...
cse_basic_bl...
cse_end_of_ba...
cse_insn
cse_main
cse_process...
cse_set_arou...
cse_rtx_vari...
create_basic...
create_delay...
```

```
base_alias_check
bitmap_clear_bi...
bitmap_copy
bitmap_bit_p
bitmap_equal_p
bitmap_element...
bitmap_find_bit
bitmap_initiali...
bitmap_operatio...
bitmap_set_bit
blocks_nrevers...
block_label
ehl_hash
emit_insns...
emit_insn_b...
emit_note_af...
emit_note_be...
emit_barrie...
emit_label_a...
emit_label_b...
emit_jump_in...
eliminate_p...
end_alias_an...
end_sequenc...
equiv_const...
eval
exact_log2...
expand...
expung...
expect...
```

```
exp_equ...
distribute_no...
diddle_return...
duplicate_loop...
dead_or_set_p
dead_or_set_re...
decl_for_comp...
delete_dead_j...
delete_insn_c...
delete_noop_m...
delete_rela...
delete_reg_e...
delete_unrea...
delete_trivi...
do_SUBST_INT
get_addr
get_alias_set_en...
get_cse_reg_info
get_insns
get_max_uid
get_last_value_v...
get_label_after
get_related_valu...
general_operan...
gen_sequence
gen_raw_REG
gen_reg_rtx
gen_rtx_CONST...
gen_rtx_REG
gen_rtx_SUBRE...
```

```
gen_rtx_fmt_i...
gen_rtx_fmt_E
gen_rtx_fmt_e...
gen_rtx_fmt_u...
gen_rtx_fmt_w
gen_jump
gen_label_rtx
gen_lowpart_if...
gen_lowpart_co...
gen_lowpart_fo...
ggc_add_rtx_root
ggc_add_root
ggc_alloc
ggc_push_context
ggc_pop_context
ggc_collect
ggc_del_root
fixed_scalar...
first_insn_af...
find_compari...
find_basic...
find_base_t...
find_base_v...
find_best_ad...
find_empty_s...
find_evalua...
find_single_u...
find_reg_not...
find_regno_p...
find_regno_n...
```

```
find_unreach...
free_basic_blo...
free_bb_for_ins...
free_EXPR_LIST_l...
free_dom_info
free_INSN_LIST_l...
free_propagate...
free_edge
flow_delete_b...
flow_dfs_comp...
floor_log2_wid...
force_to_mod...
forwarder_b...
for_each_rtx
fold_rtx
ix86_address_co...
ix86_compariso...
ix86_binary_ope...
ix86_decompose...
ix86_find_base_t...
ix86_hard_regno...
initiali...
init_alia...
init_prop...
init_reg_l...
init_dom_i...
init_labe...
inside...
insert...
instan...
```

```
insn_in...
insn_li...
int_size_in...
in_expr_lis...
invalida...
invert_br...
invert_ex...
invert_ju...
immediate_oper...
if_then_else_con...
higher_prime_num...
htab_create
htab_expand
htab_traverse
htab_find_slot_wi...
host_integerp
hook_void_bool_f...
maybe_remove...
max_reg_num
make_compoun...
make_eh_edg...
make_edges
make_extra...
make_field_as...
make_insn_raw
make_jump_ins...
make_label_ed...
make_new_qty
make_regs_eqv
mark_all_labe...
```

```
mark_regs_liv...
mark_jump_lab...
mark_used_reg...
mark_set_1
mark_set_regs
merge_blocks_n...
merge_equiv_cl...
mems_in_disj...
memrefs_con...
memory_addr...
memory_oper...
mention_regs
mode_for_size
move_deaths
label_is_jump_ta...
link_roots
life_analysis
legitimate_add...
lookup_as_funct...
lookup_for_remo...
open_dump_file
optimize_siblin...
onlyjump_p
num_sign_bit_copi...
num_validated_ch...
next_active_in...
next_real_insn
next_nonnote_i...
new_basic_bloc...
new_elt_loc_lis...
```

never_reached...

notice_stac...

notreg_cost

note_stores

noop_move_p

nonimmedi...

nonzero_bi...

nonmemory...

nonoverla...

page_group_in...

pc_set

phi_alternat...

plus_constan...

prev_real...

prev_nonn...

predict_i...

preferra...

propagate_b...

propagate_o...

push_operan...

purge_all_de...

purge_dead_e...

purge_line_n...

safe_has...

sbitmap...

sequen...

setup...

set_pa...

set_bl...

set_un...

```
set_li...
set_no...
simpl...
side_e...
singl...
shallow...
specqs...
splay_t...
ssa_rena...
ssa_elim...
subst
subreg...
start_se...
swap_co...
re...
rtx_alloc
rtx_cost
rtx_equal_p
rtx_equal_for_mem...
rtx_varies_p
rtvec_alloc
update_table_tic...
update_br_prob_no...
update_life_info
update_forwarde...
uses_addressof
use_related_val...
unshare_all_rtl_1
unshare_all_decl...
tail_recursion_l...
```

```
timevar_push
timevar_pop
tidy_fallthru_e...
try_combine
try_redirect_b...
try_simplify_c...
try_optimize_c...
try_forward_ed...
true_depende...
trunc_int_for...
tree_low_cst
varray_init
varray_grow
validate_change
x86_64_sign_exte...
x86_64_zero_exte...
x86_64_movabs_op...
x86_64_general_o...
xcalloc
xrealloc
xmalloc
```

F.6 403.gcc -O2 Functions in Selected Regions for TLP

```
adjust_offs...
address_co...
addr_affec...
addr_side_e...
aliases_ev...
alias_sets...
allocate_r...
```

```
alloc_page
any_condjump...
any_uncondju...
approx_reg_c...
apply_chang...
asm_noperand...
avoid_consta...
cached_make...
cancel_ch...
canon_has...
canon_re...
canon_rt...
can_throw...
change_addr...
check_depen...
check_for_la...
close_dump_fi...
copy_rtx
compari...
commuta...
count_reg...
count_bas...
const0...
const_i...
control...
cse_around_lo...
cse_basic_blo...
cse_end_of_bas...
cse_insn
cse_main
```

```
cse_process_n...
cse_set_aroun...
cse_rtx_varie...
create_basic...
base_alias_check
bitmap_initializ...
bitmap_clear
bitmap_element_al...
bitmap_element_li...
bitmap_set_bit
bitmap_find_bit
equiv_constant
exact_log2_wide
expand_mult_add
expand_binop
expand_expr
exp_equiv_p
decl_for_componen...
delete_reg_equiv
delete_trivially...
get_addr
get_alias_set_en...
get_insns
get_cse_reg_info
get_related_valu...
general_operan...
gen_raw_REG
gen_rtx_CONST_I...
gen_rtx_REG
gen_rtx_SUBREG
```

```
gen_rtx_fmt_i0
gen_rtx_fmt_ei
gen_rtx_fmt_ee
gen_lowpart_if_p...
gen_lowpart_com...
ggc_alloc
ggc_collect
fixed_scalar_an...
first_rtl_op
find_compariso...
find_basic_bl...
find_base_ter...
find_base_val...
find_best_addr
find_reg_note
for_each_rtx
fold_rtx
ix86_address_cos...
ix86_expand_bina...
ix86_decompose_a...
ix86_find_base_te...
immed_double_con...
init_alias_an...
inside_bas...
insert_reg...
insn_inval...
integer_zero...
integer_onep
invalidate_m...
invalidate_f...
```

host_integerp

max_reg_num

make_regs_eqv

make_edges

make_new_qty

merge_equiv_cla...

mems_in_disjo...

memrefs_conf...

memory_addre...

memory_opera...

mention_regs

legitimate_addre...

loop_optimize

lookup_as_functi...

lookup_for_remov...

open_dump_file

next_real_insn

next_nonnote_ins...

new_basic_block

notreg_cost

note_stores

nonimmediate_o...

nonoverlappin...

pc_set

prev_nonnote_in...

preferrable

push_operand

safe_hash

scan_loop

set_page_tab...

```
set_page_gro...
set_unique_r...
simplify...
side_effe...
single_se...
specqsort
splay_tre...
split_tre...
subreg_lowp...
swap_commu...
swap_condi...
record_s...
record_j...
recog_1...
recog_2
recog_5
registe...
reg_scan...
reg_ment...
reg_over...
refers_to...
rehash_us...
remove_in...
remove_fr...
replace_e...
rest_of_co...
reverse...
rtx_alloc
rtx_cost
rtx_equal_p
```

```
rtx_equal_for_memr...
rtx_varies_p
use_related_value
timevar_push
timevar_pop
tidy_fallthru_ed...
true_dependenc...
trunc_int_for_mo...
tree_low_cst
varray_init
validate_change
x86_64_sign_exten...
x86_64_zero_exten...
xcalloc
xmalloc
```

F.7 445.gobmk -O0 Functions in Selected Regions for TLP

```
approxlib
accuratelib
assimilate_st...
assimilate_ne...
add_stone
attack1
attack3
attack2
attack4
chainlinks2
create_new_stri...
clear_board
clear_move_reas...
```

```
countstones
countlib
count_common...
break_chain3_mo...
break_chain2_mo...
break_chain2_ef...
break_chain2_de...
break_chain_mov...
buildSpiralOrde...
extend_neighbor_s...
edge_clamp_moves
edge_closing_bac...
edge_block_moves
draw_back
defend1
defend3
defend2
defend4
double_atari_ch...
do_attack
do_get_read_resu...
do_find_superst...
do_find_break_ch...
do_find_defense
do_play_move
do_remove_strin...
do_trymove
gameinfo_pla...
gameinfo_cle...
gameinfo_loa...
```

```
gametree
get_read_resul...
get_moveY
get_moveXY
gtp_attack
gtp_decode_co...
gtp_defend
gtp_finish_res...
gtp_internal_s...
gtp_main_loop
gtp_mprintf
gtp_loadsgf
gtp_print_cod...
gtp_print_ver...
gtp_printf
gtp_start_resp...
gg_srand
gg_drand
gnugo_add_ston...
gnugo_clear_bo...
gnugo_set_komi
fastlib
findstones
findlib
find_cap2
find_defense
find_liberties...
find_origin
find_persisten...
find_superstri...
```

```
is_ko
is_edge_vertex
is_suicide
is_self_atari
iterate_tgfsr
incremental_or...
inv_rotate
hashtable_cl...
hashtable_en...
hashtable_se...
hashdata_inv...
hashdata_rec...
hashnode_sea...
hashnode_new...
has_neighbor
have_common_lib
hane_rescue_move...
komaster_trymove
mark_string
match
liberty_of_string
order_moves
nexttoken
next_rand
new_position
node
propagate_st...
propident
property
proper_supe...
```

```
propose_edge...
propvalue
popgo
same_strin...
search_p...
sequenc...
set_up_sn...
set_dept...
sgfNe...
sgfMk...
sgftr...
sgfGe...
sgfFr...
simple_lad...
sort_moves
special_at...
special_re...
superstri...
store_pers...
reading_cache...
readsgffile
reset_move_hist...
reset_engine
remove_liberty
remove_neighbo...
rotate_on_input
rotate_on_output
update_liberties
undo_trymove
transformation_i...
```

```
tryko
trymove
xalloc
```

F.8 445.gobmk -O2 Functions in Selected Regions for TLP

```
approxlib
accuratelib
assimilate_st...
assimilate_ne...
add_stone
attack1
attack3
attack2
attack4
chainlinks2
create_new_strin...
countstones
countlib
count_common_l...
break_chain2_move...
break_chain2_effi...
break_chain_moves
extend_neighbor_s...
edge_clamp_moves
edge_closing_bac...
edge_block_moves
draw_back
defend1
defend3
defend2
```

```
double_atari_ch...
do_attack
do_get_read_resu...
do_find_superst...
do_find_break_ch...
do_find_defense
do_play_move
do_remove_strin...
do_trymove
gameinfo_play_sg...
get_read_result
gtp_printf
fastlib
findstones
findlib
find_persisten...
find_cap2
find_superstri...
find_liberties...
find_origin
is_ko
is_edge_vertex
is_suicide
is_self_atari
incremental_orde...
hashtable_clea...
hashtable_ente...
hashtable_sear...
hashdata_inver...
hashnode_searc...
```

```
hashnode_new_re...
hane_rescue_moves
komaster_trymove
mark_string
liberty_of_string
order_moves
new_position
propagate_strin...
propose_edge_mov...
popgo
same_string
set_up_snapba...
sgftreeForw...
special_atta...
special_resc...
superstring...
store_persis...
reset_move_histor...
remove_liberty
remove_neighbor
update_liberties
undo_trymove
tryko
trymove
```

F.9 458.sjeng -O0 Functions in Selected Regions for TLP

```
Bishop
King
Knight
QProbeTT
```

```
QStoreTT
Queen
Pawn
ProbeTT
StoreTT
Rook
add_capture
add_move
checkECache
check_legal
comp_to_san
bishop_mobility
eval
gen
findlowest
f_in_check
is_attacked
is_draw
initialize_eva...
interrupt
in_check
hash_extract_pv
make
order_moves
qsearch
push_pawn_simple
push_king_castle
push_knighT
push_slidE
post_thinking
```

```
post_fail_thinki...
post_fl_thinking
search_root
see
setup_attacker...
stringize_pv
std_eval
storeECache
remove_one
rdifftime
rook_mobility
rtime
unmake
think
```

F.10 458.sjeng -O2 Functions in Selected Regions for TLP

```
hash_extract_pv
std_eval
push_king
push_knighT
push_slidE
```

F.11 462.libquantum -O0 Functions in Selected Regions for TLP

```
add_mod_n
addn_inv
main
madd_inv
muxha_inv
muxfa_inv
```

```
mul_mod_n
muln_inv
emul
quantum_addscrat...
quantum_add_hash
quantum_cnot
quantum_exp_mod_n
quantum_decohere
quantum_delete_ma...
quantum_gate1
quantum_gate_cou...
quantum_get_state
quantum_imag
quantum_hash64
quantum_hadamard
quantum_memman
quantum_objcode_pu...
quantum_new_matrix
quantum_qec_get_sta...
quantum_prob_inlin...
quantum_sigma_x
quantum_swapthel...
quantum_real
quantum_toffoli
test_sum
```

F.12 462.libquantum -O2 Functions in Selected Regions for TLP

```
quantum_bmeasure
quantum_frand
quantum_memman
```

quantum_objcode_pu...

quantum_state_coll...

quantum_real

spec_rand

main

F.13 471.omnetpp -O2 Functions in Selected Regions for TLP

cMessage30

cObject

EtherAppReq

EtherAppResp

EtherCtrl

EtherFrameWith...

genk_dblrand

MACAddress_Base

operator<=

operator>

_Z11expon...

_Z11trunc...

_Z10intun...

_Z10opp_st...

_Z12opp_ne...

_Z12genk_i...

_Z14check_a...

_Znwm

_ZNK...

_ZN1...

_ZNS...

_ZN2...

_ZN5...

```
_ZN4...
_ZN7...
_ZN6...
_ZN9...
_ZN8...
_ZdlPv
_Z6normalddi
_Rb_tree_iterator
_wrap_intuniform
_wrap_exponential
_wrap_truncnormal
~MACAddress_Base
~cPolymorphic
~cMessage30
~cObject
~EtherAppReq
~EtherAppResp
~EtherCtrl
~EtherFrameWith...
```

F.14 473.astar -O0 Functions in Selected Regions for TLP

```
pointt
regobj
_Z13_aligned_f...
_Z15_aligned_m...
_Z8myrandomv
_Z3sqri
_Z7random1i
_Z7myroundf
_ZN11regbo...
```

```
_ZN15large...
_ZN9stati...
_ZN9regmn...
_ZN9flex...
_ZN6regobj...
_ZN6wayobj...
```

F.15 473.astar -O2 Functions in Selected Regions for TLP

```
pointt
regobj
_Z13_aligned_f...
_Z15_aligned_m...
_Z8myrandomv
_Z3sqri
_Z7random1i
_Z7myroundf
_ZN11regbo...
_ZN15large...
_ZN9stati...
_ZN9regmn...
_ZN9flex...
_ZN6regobj...
_ZN6wayobj...
```

F.16 483.xalancbmk -O0 Functions in Selected Regions for TLP

```
AbstractSt...
AbstractNu...
AllContentM...
AnySimpleTy...
```

ArrayInde...

ArrayJani...

ArenaAllo...

ArenaBloc...

AttrImpl

Attribut...

AttrMapI...

AttrNSIm...

AVTPrefixCh...

CurrentNodePus...

CMStateSet

CMLeaf

CountersTabl...

CommitPushPa...

ConcatToken

Context

ContentSpe...

BaseRefVect...

Base64Binar...

BitSet

BinMemOutp...

BinFileIn...

BinFileOu...

BMPattern

BuildWrapper...

BorrowRetur...

BooleanData...

EmptyStackEx...

ExtensionNS...

ExtensionFu...

EEndianNameM...

Elem...

ENameMapFor

DoubleToDOMS...

DeleteFunct...

DefaultColl...

DOMA...

DOMC...

DOMB...

DOME...

DOMD...

DOMN...

DOMP...

DOMS...

DOM_N...

DGXMLScanner

DFAContentMo...

GuardCachedObj...

GrammarResolve...

GetAndReleaseC...

FieldMatcher

Function...

FormatterTo...

IconvTrans...

IDREFData...

IDDatatyp...

IC_KeyRef

IC_Unique

IOExceptio...

IllegalArg...

InputSo...

InScope...

InMemMs...

Invalid...

IdentityCo...

KeyDeclaration

KeyTable

Janitor

MalformedURL...

MatchPattern...

MixedContentM...

MemBufFormatT...

ModifierToke...

MonthDayDat...

MonthDataty...

LastPoppedHolde...

LongToDOMString

OutputContextSt...

OutputString

NameIdPool

NameSpace

Namespace

NCNameDataty...

NumberForma...

NullPointer...

NoSuchEleme...

NodeSortKe...

NodeRefLis...

NOTATIONData...

QName

```
ParamsVectorE...
ParseExceptio...
PrefixResolve...
ProcessingIns...
PushAndPopCo...
PushAndPopEl...
PushPopInclud...
SAXParser
SAXParseE...
SAXExcepti...
SAXNotSup...
SAXNotRec...
SchemaAttD...
SchemaElem...
SchemaGram...
SelectionEv...
SetAndResto...
StackEntry
Stylesheet...
SGXMLScanner
RuntimeExceptio...
ReaderMgr
ResultNames...
ReusableAre...
RegxParser
RegularExp...
RefHash2...
RefHashT...
RefCounte...
RefVector...
```

URLInputSource

UTFDataFormatE...

UnionDatatyp...

UnsignedLong...

UnexpectedEO...

ThrowEOEJanito...

TracerEvent

TraverseSche...

Transcoding...

TranscodeT...

TranscodeF...

TopLevelArg

Wrapper4InputSo...

Wrapper4DOMInpu...

VariablesStack

ValueStoreCach...

ValueVectorOf

Xalan...

Xerces...

XML...

XObjectI...

XObjectP...

XPathMat...

XPathExc...

XS...

XResultTr...

XToken

_Construct<xalanc_1...

_Construct<std::vect...

_Bit_iterator_base

```
_Bit_const_iterato...

_Bvector_impl

_Deque_impl

_Deque_iterator

_Deque_base

_Destroy<xalanc_1_8...

_Destroy<std::deque...

_Destroy<std::_Deque...

_Destroy<std::vecto...

_Destroy<__gnu_cxx::__no...

_GLOBAL__I__Z5Usagev

_GLOBAL__I__ZN11xer...

_GLOBAL__I__ZN10xal...

_M_allocate_and_copy...

_M_insert_dispatch<c...

_M_insert_dispatch<__g...

_M_range_initialize...

_Rb_tree_impl

_Rb_tree_iterator

_Rb_tree_const_itera...

_Vector_impl

_Vector_base

_ZN11x...

_ZN10x...

_ZN12I...

_ZN9__gnu_c...

_ZNK1...

_ZNK9...

_ZNKS...

_ZNK6...
```

```
_ZNSt...

_ZNSols...

_ZStm...

_ZSt1...

_ZSt2...

_ZSt4...

_ZSt6...

_ZSt9...

_ZSt8...

_Z5Usagev

_Z41__static_i...

_Z7stricmp...

_Z7getArgs...

_Z8xsltMa...

_Z8strnic...

_Z8getXPa...

_ZThn1...

_ZThn3...

_ZThn2...

_ZThn4...

_ZThn6...

_ZThn8...

__advance<std::_Deque_i...

__copy_b<xalanc_1_8::Ar...

__copy_b<xalanc_1_8::At...

__copy_b<xalanc_1_8::Ele...

__copy_b<xalanc_1_8::Nod...

__copy_b<xalanc_1_8::Reu...

__copy_b<xalanc_1_8::Var...

__copy_b<xalanc_1_8::Xa...
```

```
__copy_b<xalanc_1_8::Xe...
__copy_b<char>
__copy_b<const xala...
__copy_b<short unsi...
__copy_b<std::_Deque_it...
__copy_b<std::vector<x...
__copy_b<std::vector<s...
__copy_b<std::vector<d...
__distance<const ...
__destroy_aux<xala...
__destroy_aux<std::s...
__destroy_aux<std::d...
__destroy_aux<std::_D...
__destroy_aux<std::v...
__do_global_dtors_a...
__lg<long int>
__normal_iterator
__static_initializa...
__uninitialized_cop...
__uninitialized_fil...
__tcf_111
__tcf_113
__tcf_117
__tcf_109
__tcf_105
__tcf_107
__tcf_18
__tcf_0
__tcf_38
__tcf_29
```

```
__tcf_27
__tcf_51
__tcf_59
__tcf_48
__tcf_43
__tcf_72
__tcf_75
__tcf_63
__tcf_64
__tcf_91
__tcf_92
__tcf_95
__tcf_80
__tcf_88
appendBtoFLis...
append<xalanc_1...
assign
addIfNotFound
castToChild...
call_gmon_sta...
charAt
clear
c_str
c_wstr
copy<short...
copy<std::re...
copy<std::de...
copy<std::_Bi...
copy<std::_De...
copy<char>
```

copy<const...

copy<xalanc...

copy<double...

copy<long u...

copy_backwa...

compareIgno...

equalsIgnoreCase...

distance<__gnu_cxx...

deque

doAppendCh...

doAppendSi...

doAppendTo...

doCollationC...

doWarn

getAtt...

getChi...

getEx...

getEv...

getDoc

getKey...

getNum...

getPar...

getSu...

getSt...

getTra...

getXPa...

gValidatorMutex

fill<xalanc_1_8::C...

fill<xalanc_1_8::N...

fill<const voi...

```
fill<short un...
fill<std::deque...
fill<std::vecto...
fill_n<const vo...
fill_n<short un...
find<std::_Deque_i...
find<__gnu_cxx::__nor...
find_if<__gnu_cxx::__n...
frame_dummy
isPrefixUsed...
isPendingAtt...
isEmpty
isXMLLetterOr...
indexOf
map
max<size_t>
length
operator ne...
operator+
operator-<xa...
operator-<co...
operator-<st...
operator==<x...
operator<< <st...
operator>>
pair
startsWith
swap<xalanc_1_8::Att...
swap<xalanc_1_8::Xal...
swap<xalanc_1_8::Key...
```

swap<xalanc_1_8::Na...

swap<xalanc_1_8::No...

swap<size_t>

swap<short uns...

swap<std::deque<x...

swap<std::_Bit_ite...

swap<std::_Bit_typ...

swap<std::_Deque_i...

swap<std::vector...

swap<unsigned in...

swap<long unsign...

releaseMemory

returnExternalM...

uninitialized_fil...

transcodeStrin...

transform

toCharArray

vector

~AbstractNum...

~AllContentM...

~AnySimpleTy...

~ArrayJanit...

~ArenaAllo...

~ArenaBloc...

~AttrImpl

~Attribute...

~AttrMapIm...

~AVTPartXPa...

~AVTPrefixC...

~CurrentNodePus...

~CMStateSet

~CMLeaf

~CMNode

~CommitPushPa...

~CollationCo...

~CollectionD...

~ContextNode...

~ContentHan...

~ContentSpe...

~BaseRefVector...

~BinMemOutput...

~BinOutputStr...

~BuildWrapperT...

~BorrowReturn...

~EEndianName...

~Elem...

~EnsurePo...

~EnsureDe...

~EndOfEnti...

~EntityRes...

~ErrorHandle...

~ExtensionN...

~ExtensionF...

~ENameMapFor

~DocTypeHandl...

~DeclHandler

~DefaultHand...

~DTDHandler

~DOMA...

~DOMC...

```
~DOMB...

~DOMD...

~DOMI...

~DOMN...

~DOMP...

~DOMS...

~DOMX...

~DFAContentMo...

~GetAndReleaseCac...

~FunctionID

~FunctionCu...

~FunctionCo...

~FunctionS...

~FunctionDif...

~FunctionGen...

~FormatterStrin...

~FormatterTree...

~FormatterToX...

~FormatterToN...

~IconvTransSe...

~IC_Selector

~IC_Unique

~IDREFDatatyp...

~IGXMLScanner

~InMemHandl...

~InMemMsgLo...

~InvalidQNa...

~InvalidSta...

~HashCMStateSet

~HashBase
```

```
~Janitor
~MalformedURLE...
~MixedContentM...
~MemBufFormatT...
~ModifierToken
~LastPoppedHold...
~LexicalHandler
~LocalFileForm...
~Locator
~OutputContextPu...
~OutputContextSt...
~NCNameDatatypeV...
~NodeSortKey
~NodeSorter
~Parser
~PSVIAttribut...
~PrefixResolv...
~PushAndPop...
~PushPopIncl...
~PopAndPushCo...
~SAXParser
~SAXParseEx...
~SAXExceptio...
~SchemaAtt...
~SchemaEle...
~SchemaGra...
~Scope
~SelectorMat...
~SetAndResto...
~Stylesheet...
```

```
~SGXMLScanner
~ReaderMgr
~ResolveAndCl...
~ReusableAren...
~RegxParser
~RegularExpr...
~RefHash2Ke...
~RefHashTab...
~RefVectorOf
~URLInputSource
~UnknownEncodi...
~UnexpectedEOF...
~UnsupportedEn...
~TracerEvent
~TraceListener...
~TranscodingEx...
~TranscoderInt...
~TopLevelArg
~WFXMLScanner
~VariablesStack
~ValueVectorOf
~Xalan...
~Xerces...
~XML...
~XObjectP...
~XNodeSet
~XPathMat...
~XS...
~XResultT...
~XToken
```

```
~_Vector_base
~_Rb_tree
~_Deque_base
~_List_base
~deque
~map
~pair
~vector
```

F.17 483.xalancbmk -O2 Functions in Selected Regions for TLP

```
ArenaBlock
CurrentNodePush...
CommitPushPara...
ContextNodeLis...
BorrowReturnNo...
BorrowReturnMu...
BorrowReturnFo...
ElementPropertie...
DOMStringPrintWr...
GetAndReleaseCac...
FormatterString...
FormatterListen...
MutableNodeRefLi...
OutputContextPus...
NumberFormatStr...
NodeSortKey
NodeRefListBa...
NodeTester
ParamsPushPo...
ParamsVector...
```

```
PrintWriter
PrefixResolv...
PushAndPopC...
PushAndPopEl...
PopAndPushCon...
SetAndRestoreC...
StackEntry
ReusableArenaBlo...
Writer
VectorEntry
Xalan...
XObjectPtr
XObjectRes...
XNumberBase
XStringRe...
XStringCa...
XStringBa...
XResultTree...
XTokenStri...
XTokenNumb...
_Construct<xalanc_1...
_Construct<std::vect...
_Bit_const_iterator
_Deque_iterator
_Destroy<xalanc_1_8...
_Destroy<std::vecto...
_M_insert_dispatch<c...
_M_insert_dispatch<__g...
_Temporary_buffer
_Vector_impl
```

```
_Vector_base

_ZSt16__ins...

_ZSt16__mer...

_ZSt2...

_ZSt5merg...

_ZSt4fill...

_ZSt6__fin...

_ZSt6fil...

_ZSt9__find...

_ZSt8for_e...

_ZThn8_NK10xala...

_ZThn64_NK10xal...

_ZN11xerces...

_ZN10xalanc...

_ZN9__gnu_cxx1...

_ZNK10xala...

_ZNK9__gnu_cx...

_ZNKSt1...

_ZNKSt8...

_ZNKSt3...

_ZNKSt6...

_ZNSt1...

_ZNSt8_Rb_t...

_ZNSt5deq...

_ZNSt6vec...

__advance<const xal...

__copy_b<xalanc_1_8::Nod...

__uninitialized_cop...

__uninitialized_fil...

__distance<const x...
```

```
__destroy_aux<xalan...
__destroy_aux<std::ve...
__normal_iterator
append
assign
advance<const x...
charAt
createEmptyS...
clear
c_wstr
copy<xala...
copy<cons...
copy<shor...
copy_back...
compareIg...
collation...
consume...
convertH...
equalsIgnoreCase...
doAppendChildN...
doCollationCom...
doValidate
getChildren...
getChildDat...
getResult
getSingleT...
getSubstri...
getStartI...
getString...
get_temporary...
```

```
getTotal
fill<short uns...
fill<std::vector...
fill_n<short uns...
find<__gnu_cxx::__norm...
find_if<__gnu_cxx::__no...
isXMLLetterOr...
isEmpty
isNamespaceDe...
indexOf
max<size_t>
length
operator==
operator-<xalanc...
operator-<const ...
operator-<short ...
notCached
startsWith
stable_sort<__gnu_c...
swap<xalanc_1_8::Xal...
reserve
uninitialized_fil...
toCharArray
vector
~CurrentNodePush...
~CommitPushPar...
~CollectionCle...
~ContextNodeLi...
~BorrowReturnNod...
~BorrowReturnFor...
```

```
~EnsurePop
~DOMStringPrintWr...
~GetAndReleaseCac...
~FormatterString...
~FormatterListen...
~OutputContextPus...
~NodeSortKey
~PrintWriter
~PrefixResolve...
~PushAndPopCont...
~PopAndPushCont...
~SetAndRestoreCu...
~SetAndRestoreCo...
~Writer
~vector
~Xalan...
~XObjectPtr
~XObjectRes...
~XNumberBase
~XStringRe...
~XStringCa...
~XStringBa...
~XResultTree...
~XTokenStri...
~XTokenNumb...
~_Temporary_buffer
~_Vector_base
```