UC Santa Cruz

UC Santa Cruz Electronic Theses and Dissertations

Title

Section Based Program Analysis to Reduce Overhead of Detecting Unsynchronized Thread Communication

Permalink

https://escholarship.org/uc/item/8vx8d8wv

Author

Das, Madan Mohan

Publication Date

2015

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

SECTION BASED PROGRAM ANALYSIS TO REDUCE OVERHEAD OF DETECTING UNSYNCHRONIZED THREAD COMMUNICATION

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Madan Mohan Das

March 2015

The Dissertation of
Madan Mohan Das is approved:

Professor Jose Renau, Chair

Professor Cormac Flanagan

Professor Anujan Varma

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Copyright © by

Madan Mohan Das

2015

Table of Contents

List of Figures **v**

List of Tables **vii**

Abstract **viii**

Acknowledgments **ix**

1 Introduction **1**
1.1 Contributions 8
1.2 Thesis Organization 10

2 Related Work **12**
2.1 Static Race Detection 13
2.2 Dynamic Race Detection 15
2.3 Deterministic Runtime Systems 18
2.4 Software Transactional Memory (STM) 19
2.5 Data Flow Analysis 21
2.6 Pointer Analysis Background 22
2.6.1 Flow sensitive vs. insensitive pointer analysis 23
2.6.2 Context sensitive vs. insensitive pointer analysis 24
2.6.3 Dynamic vs. Static pointer analysis 25
2.7 Some important Flow insensitive methods 26
2.7.1 Andersen's flow insensitive analysis 26
2.7.2 Steensgaard's algorithm 27

3 Finding Disjoint Thread Sections **29**
3.1 Terminologies 29
3.2 SBPA Pointer Analysis Framework 32
3.2.1 Perform modref analysis per section 33
3.2.2 Adaptive non-unification for points-to sets of function arguments . . . 33

3.2.3 Field sensitivity for array elements . . . 36
3.3 Constructing the Reduced ICFG . . . 36
3.4 Single-Threaded Thread Sections (Single-TS) . . . 41
3.5 Disjoint Thread Sections (Disjoint-TS) . . . 45
3.6 Overall Instrumentation Flow . . . 50

4 Programmer Annotations and MTROM 51
4.1 Marking Parallel Code Sections . . . 52
4.2 Multi Thread Read Only Memory . . . 52

5 Loop Invariant Log Motion 56
5.1 Scalar Loop Invariant Log Motion (SLILM) . . . 57
5.2 Vector Loop Invariant Log Motion (VLILM) . . . 58
5.3 Result of Applying LILM . . . 61

6 Experimental Results 63
6.1 Experimental Setup . . . 63
6.2 Results . . . 65
6.2.1 Overall Results . . . 66
6.2.2 Analysis of Reduction in Instrumentation . . . 68
6.2.3 Benchmark Insights . . . 70
6.2.4 Compilation Overhead . . . 75

7 A case study with ThreadSanitizer 78

8 Improving Static Race Precision with SBPA 84
8.1 Methodology . . . 85
8.2 An Example Case . . . 87
8.3 Results . . . 91

9 Conclusion 96

10 Future Work 98
10.1 Symbolic Array Partitioning . . . 98
10.2 Generalized SBPA and MTROM . . . 99
10.2.1 Extension of SBPA to Tree of Threads . . . 99
10.2.2 MTROM for Tree of Threads and Dynamic MTROM . . . 100
10.3 Hardware Implementation of MTROM . . . 101

Bibliography 104

List of Figures

1.1 Two threads T1 and T2 execute four different code sections (CS1, CS2, CS3, CS4) separated by a barrier B. The barrier implicitly creates two thread sections (TS1, TS2) that cannot execute simultaneously. W_X and R_X represent write and read of some memory location X. Since X is not modified anywhere in TS2, R_X does not need to be instrumented. 6
1.2 A typical threaded section of program 7

3.1 A code snippet showing the call of a function f, that has any of the thread creation, join or synchronization directive. The resulting changes in RICFG are shown on the right. 41
3.2 A program comprised of 4 code sections in 3 thread sections 42
3.3 A decomposition of threaded section into smaller, disjoint parallel segments . . 46
3.4 Two threads executing 5 code sections in 3 disjoint thread sections. 47

6.1 SBPA compiler pass detected 63% of all memory accesses at run-time as non-conflicting. Excluding the improvements from 'Directives' yields 51% accesses proven as non-conflicting. 66
6.2 SBPA identified 80% of the dynamic memory accesses executed in single threaded mode. 67
6.3 68% of the loads are detected as non-conflicting, with a few applications reaching 100%. 68
6.4 61% of the stores do not need to be tracked by tools like data-race detectors which is much better than 38% with Base. For STMs that may have restarts, 52% of the stores can be proven as safe. 69
6.5 Compilation times of Base and SBPA normalized against compile time when no optimization is applied. On average, Base took 75% and SBPA took 46% of unoptimized compile time. 76

7.1 SBPA yields over 2 times speedup compared to Tsan, with some applications achieving over 30 times speedup. 82

7.2 ThreadSanitizer speed-ups in the different modes described earlier. SBPA, combined with Directives, speeds up ThreadSanitizer execution by a factor of 2.74. 82

8.1 Static race detection flow with SBPA. Static races detected are classified and further validated by dynamic race detection. 88
8.2 Percentages of total read and write lines identified as non-racy by Base and SBPA techniques. 93
8.3 Percentages of read lines identified as non-racy by Base and SBPA techniques. . 93
8.4 Percentages of write lines identified as non-racy by Base and SBPA techniques. 93
8.5 Percentages of read and write instructions identified as non-racy by Base and SBPA techniques. 94
8.6 Percentages of read instructions identified as non-racy by Base and SBPA techniques. 94
8.7 Percentages of write instructions identified as non-racy by Base and SBPA techniques. 94

10.1 A hierarchy of threads in a program, represented as a tree 100

List of Tables

5.1 Total access logging reduction improvement with LILM 62
5.2 Load logging reduction with LILM. 62

6.1 Benchmarks used, abbreviation shown in parentheses. 64

7.1 Runtimes (in seconds) of executables compiled for ThreadSanitizer dynamic race detection with 2 threads. 80

8.1 Total racy read and write lines, and total program lines for the benchmarks studied. 95

Abstract

Section Based Program Analysis to Reduce Overhead of Detecting Unsynchronized Thread Communication

by

Madan Mohan Das

Most systems that test and verify parallel programming, such as data race detectors and software transactional memory systems, require instrumenting loads and stores in an application. This can cause a very significant runtime and memory overhead compared to executing uninstrumented code. Multithreaded programming typically allows any thread to perform loads and stores at any location in the process's address space independently. Most of these unsynchronized memory accesses are non-conflicting in nature; that is, the values read from or written to memory are only used by a single thread. We propose Section-Based Program Analysis (SBPA), a novel way to decompose the program into disjoint sections to identify non-conflicting loads and stores during program compilation.

We combine SBPA with improved context sensitive alias analysis, loop specific optimizations and a few user directives to further increase the effectiveness of SBPA. We implemented SBPA for a deterministic execution runtime environment, and were able to eliminate 63% of dynamic memory access instrumentations. We also integrated SBPA with ThreadSanitizer, a state of the art dynamic race detector, and achieved a speed-up of 2.74 times on a geometric mean basis. Lastly, we show that SBPA is also effective in static race detection.

Acknowledgments

First, I sincerely thank my adviser Professor Jose Renau for his ideas, vision, constant support and feedback over the last many years on my research. Without his help I won't be able to reach this milestone in my life.

I would also like to thank Professor Cormac Flanagan and Professor Anujan Varma for accepting to review this work, and providing valuable feedback. This thesis wouldn't have been complete without their inputs.

I also thank Gabriel Southern, my fellow researcher and co-author in my publications for his great effort and support for the work presented in this thesis; and for bringing positive thoughts during times of difficulty.

I also sincerely thank other MASC lab students for their valuable inputs and feedback during the tenure of my studies.

Finally, I thank my wife, son and daughter for being supportive of my decision to pursue this degree and understanding my inability at times to carry out family obligations.

Chapter 1

Introduction

Since the beginning of the computer era, researchers have explored several parallel execution models and architectures. While significant progress has been achieved in parallelizing multiple program execution on the same system, and in distributed multi-processing where the outcome of the program depends on the solution of well-segregated sub-problems, parallel programming in the context of a single program with shared memory remains difficult to adopt for programmers. This fact is most conspicuous when we observe that even today, most programmers first think of their programs as single-threaded applications and often times develop as such, and then consider parallelizing the program as an after thought.

The reasons for such behavior are multiple. Most prominent among those is the fact that for software programmers, it is difficult to visualize the parallel execution of many different program fragments. Secondly, potentially exponential number of possible interleavings of the program fragment executions makes it almost impossible to contemplate and debug the

unpredictable bugs, or as many have said, the ”heisen-bugs”. Worse, if an untested execution interleaving of a program with such heisen-bugs occurs in applications when in critical deployment, consequences can be catastrophic. A few other major reasons why parallel programs are still not mainstream, are lack of debug framework, memory latency and lack of software applications to guide programmers in parallelizing their algorithms. Often times, it is not clear which portions of a program should be parallelized and what is the expected benefit of doing so. Except for very simple and obvious programs, parallelizing software applications remains non-trivial for developers.

As a consequence, most software algorithms have continued to evolve without the use of parallel programming, mainly riding on the exponential performance improvements seen in single processors through the use of deeper and deeper pipelining and smaller silicon geometries. However, in recent years, this path has hit a hurdle due to the increasing cost of such processors and practical limits on clock frequency scaling. While transistor density still seems to follow Moore's law, clock speed and CPU throughput are not on same track. Another big deterrent has been the power and thermal considerations, which are increasingly determining the clock frequency that can be used. Faster transistors require a lower threshold voltage, which in turn result in higher static current dissipation, thereby increasing the power consumption and temperature in the chip. These factors together have moved the focus of computer architects from performance gains in uni-processor systems back to parallel processing. Although performance of single stream instructions has continued to improve, such improvements have slowed down significantly, and this trend is likely to continue in the future years. Hence, there has been a renewed interest in multi-processing architectures and models in recent years.

There are many forms of multi-processing in vogue. In this work, we focus on shared memory multi-processing using general purpose CPUs. This is where the trend has been recently, due to a stagnation in CPU clock scaling. Most inexpensive systems already have 4 to 64 cores. However, it must be noted that even if the program has significant parallelism, applications won't scale indefinitely due to memory latency and bandwidth issues in such shared memory applications. So, to be more specific, we considered applications using anywhere from 2 to 32 cores for our studies. The uni-processor case is considered in this research solely for performance comparisons. In this work, we consider multi-threaded applications using POSIX pthread library only, though the ideas are generally extensible to other threading mechanisms.

While shared memory model makes communication among threads fast, it also gives rise to memory access dependencies among the threads, and these are not managed very well by today's processors. Typically, shared memory multithreaded applications assume implicit data sharing among threads, and hence thread synchronization has been a perpetual issue for parallel programs. No effective solution has been found to date. This leads to a need of software tools to detect and fix the associated issues.

Data-race detection [88], software transactional memory (STM) [91], and deterministic execution [11] have been proposed as ways to reduce errors associated with multithreaded programming. These techniques detect unsynchronized communications among threads and either report them as possible bugs (for data-race detection), or change the behavior of running threads (for STM and deterministic execution).

However, the use of these techniques is limited in practice by significant instrumentation overhead associated with software implementations [23]. These techniques perform run-

time checks on loads and stores that cause significant slowdown in program execution. One way to improve the performance of such systems is to decrease the cost of each dynamic check operation. For instance, FastTrack [41] improved on the performance of Djit+ [79] by reducing the cost of most runtime checks needed for data-race detection.

An alternative and complementary way to improve performance is to reduce the number of such checks. Static program analysis techniques can be combined with verifiable user directives to identify and eliminate checks on *non-conflicting* loads and stores for thread-local or read-only data. Dynamic instrumentation should ideally be used only when there is a possibility of unsynchronized communication among multiple threads; i.e, when these memory accesses may be $conflicting$ data among threads. More accesses being classified as non-conflicting result in fewer checks at runtime. In data-race free applications, conflicting loads and stores must be synchronized with barriers and/or mutual exclusion areas. If a compiler can prove that a store performed by a thread cannot overlap with any load or store performed by any other thread, then that store access does not require any runtime check.

A *non-conflicting* access is different from a *non-racy* access. We define *non-conflicting* memory accesses for a section of program execution contained within global barriers, i.e, synchronization points of all the threads. While non-racy accesses to same memory location can communicate information from one thread to other threads within the same multi-threaded section, a non-conflicting access never passes information between threads in the same thread section. In other words, non-conflicting values modified within a section are not used by other threads within the same section, either with or without synchronization. As an example, lock protected read and write accesses of same memory location by different threads within same

section are non-racy, but not non-conflicting. For systems such as deterministic execution and conflict free STM task execution, only non-conflicting accesses are safe, as non-racy accesses still can produce different results, depending on the order of lock acquisition.

Similarly, a *conflicting* access of a memory location might be either racy or non-racy, if the modified value in the same section is used by any thread that is different from the thread that modifies the value.

However, current compilers only identify a small subset of non-conflicting memory accesses. Current alias analysis also does not consider that some section of code cannot execute simultaneously with other sections of code. Existing solutions conservatively assume that any code section can overlap its execution with any other code section. Typically, when thread escape analysis is performed, it only marks a memory location as thread local or escaping (potentially shared between threads); it does not consider the context in which a memory address escapes, and how it is used.[1]

The key insight of this work is that it is possible to statically divide an application into thread sections, and that these sections can be used to reduce the number of instrumentation instructions a compiler inserts, and are executed at runtime. For example, Figure 1.1 shows a thread $T1$ performing a write to a shared memory location W_x in a code section $CS1$ and another thread $T2$ performing a read of the same location after a synchronized barrier. The barrier creates a *happens before* relationship between the two sections; so the two code sections cannot execute simultaneously. Consequently, the read is race free and does not need to be instrumented.

[1] Some escape analysis techniques consider thread context when constructing objects in Java.

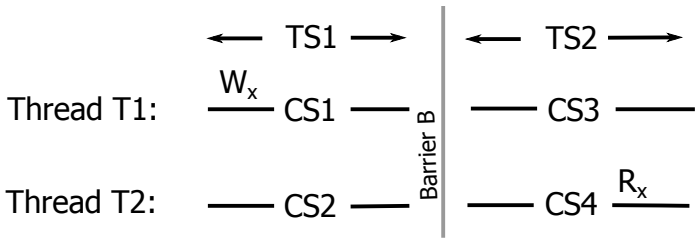

Figure 1.1: Two threads T1 and T2 execute four different code sections (CS1, CS2, CS3, CS4) separated by a barrier B. The barrier implicitly creates two thread sections (TS1, TS2) that cannot execute simultaneously. W_X and R_X represent write and read of some memory location X. Since X is not modified anywhere in TS2, R_X does not need to be instrumented.

Threaded code sections are essentially the top level code for a parallelized section of program. Since the threads created within a threaded section all terminate within the same threaded section, when we are concerned about communication pattern among threads created in this section, we only need to consider the code that is reachable from this threaded section. More precisely, it is the set of all functions that act as the start functions for the threads, and the code in parent thread that is executed in this section. Often times, the parent thread just waits for the completion of child threads or acts as a master synchronizer.

Fig 1.2 shows a typical threaded section, where three child threads are started, with function and data pairs of (F1,D1), (F2,D2) and (F2,D3). Thus, the thread starts can be seen as tuples (F_i, D_i) where function F_i is executed on data set D_i. In the shown example, S is the dominator of the thread creations, and T is the post-dominator of all thread terminations. So, a thread section can be represented as a tuple of the dominator and post-dominator instructions, as in (S, T). Then, the synchronization considerations and instrumentations can be limited to the threaded sections of a program only. This is useful since in many complex programs, the portion of program that is performance critical may be small, yet consume most of the runtime.

Identifying the threaded sections helps avoid instrumenting the loads and stores unnecessarily in other regions of the program. Thus, each thread section can be identified as a pair of source and sink vertices in the CFG.

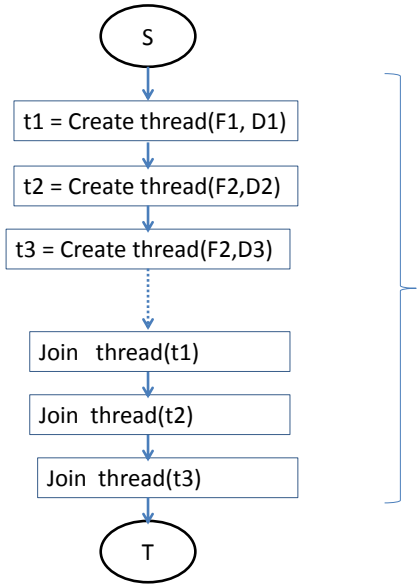

Figure 1.2: A typical threaded section of program

Previous work on data-race detection has taken advantage of the fact that loads and stores in serial sections of code do not need to be instrumented [72, 83, 84]. However, prior work mainly targeted structured programming environments such as Cilk [17] or Fortran using nested parallel loops. Structured programming explicitly restricts communication among threads, which makes it easier to determine when an application is executing a serial section of code. In contrast, this work focuses on statically partitioning an application written using unstructured parallelism with thread spawns and barriers. Our algorithms segment the whole

program into sections, and apply optimizations based on the communication pattern among sections of code that can be executed simultaneously by multiple threads. This helps it identify non-conflicting loads and stores, thereby reducing dynamic checks. To improve precision of alias analysis for memory locations accessed in these different sections, we extended flow insensitive Steensgaard [92] points-to analysis to use dynamically adjusted non-unified points-to sets. We also describe a small set of verifiable user directives that helps the compilers and programmers reason about memory accesses happening in multi-threaded mode and reduce the instrumentations further.

We integrated our techniques for use with our own implementation of a deterministic execution system similar to CoreDet [11] to accurately measure the reduction in instrumentation. We also integrated our pass with ThreadSanitizer [89], a widely used dynamic race detector, to measure runtime improvements using our optimizations. Lastly, we also measured the effectiveness of the methods in static race detection. Our methods can be combined with many different systems, such as FastTrack [41], DJIT+ [79] and others.

1.1 Contributions

In this work, we propose Section-Based Program Analysis (SBPA) with the following contributions:

- Propose a language agnostic method to statically decompose multithreaded applications into code sections, and analyze their communication pattern.
- Propose a mechanism and algorithm to identify the global barriers within multi-threaded

code sections of the program, thus increasing the number and precision of provably disjoint code sections. Prior work in the literature has focused only on identifying the multi-threaded sections. This work finds such sections, as well as the global barrier separated sections within the multi-threaded section.

- Improve precision of points-to analysis by using an adaptive non-unification strategy that limits the size of points-to sets.

- Propose loop specific optimization to reduce instrumentation in program loops.

- Propose a small set of verifiable user directives to improve static analysis

- Propose a multi-thread read only memory model (MTROM) to improve tool flow and programmer reasoning of memory accesses in parallel portions of program.

- Evaluate SBPA in a deterministic runtime system similar to CoreDet and show 51% reduction in dynamic instrumentation over varied suite of benchmarks. With few directives, the total reduction increased to 63%. In comparison, reduction by current state-of-the-art compiler techniques as used in CoreDet is only 16%.

- Integrate SBPA with ThreadSanitizer, and show reduction in execution time of instrumented programs by a factor of 2.74 on a geometric mean basis.

- Apply SBPA to the problem of static race detection and show that it is effective. We propose an integrated approach of static and dynamic race detection to develop and test multi-threaded programs.

We propose a novel way of decomposing multi-threaded programs into disjoint, barrier separated sections, which helps improve the precision of mod-ref and alias analysis, to achieve significant reduction in instrumentation overhead. Many researchers have focused on improving the precision of pointer analysis for threads. We show that it is equally important to consider the modified and referenced properties of the pointer values. A major theme of our work is to use the precision of points-to relationship in conjunction with modified-referenced information for each parallel segment of a program. Our analysis framework helps all systems focusing on race detection, determinism and debugging of concurrency related issues through rigorous compiler support for multi-threaded programs. It works for general shared memory multi-threading models.

In summary, we combine multiple techniques to create a holistic and practical solution for efficient detection and correction of race and non-determinism related issues in multi-threaded programs.

1.2 Thesis Organization

The rest of this dissertation is organized as follows. Chapter 2 surveys related work in this field. Chapter 3 describes the partitioning of an application into sections and how this is used to reduce instrumentation based on such partitioning. Chapter 4 describes a few programming directives and MTROM memory. Chapter 5 describes loop specific optimizations. Chapter 6 describes the experimental setup, and presents results and insights based on our in-house runtime library. Chapter 7 presents results by integrating our methods with ThreadSan-

itizer, an industry standard dynamic race detection software. Chapter 8 presents the results of applying our methods in reducing false positives in static race detection. Chapter 9 concludes. Chapter 10 presents future extensions possible from this work.

Chapter 2

Related Work

The methods used in this work have a solid foundation in static program analysis, a field that has matured with decades of research. Static program analysis covers multiple techniques, such as control and data flow analysis, constraint based analysis, concurrency analysis, type and effect systems and many others. These techniques can have a variety of objectives, such as code optimization, static race detection and correction checks to name a few. However, our work is applied to reduce overhead of dynamic checks. So, it is also closely related to dynamic race detection and other types of runtime checks. In this chapter, we briefly describe the prior work in all the fields that have a strong relationship with our work; and in particular we discuss the current techniques in the areas of both static and dynamic race detection. As discussed below, a vast body of literature [27, 74, 80, 96] exists, focusing on both static and dynamic race detection and other program analyses.

2.1 Static Race Detection

Naik et al. [74] described `Chord`, a tool to statically detect races in Java programs, that uses many techniques similar to SBPA analysis methods, such as identification of thread local variables and lock protected accesses. Chord uses an optimization *Unlocked Pairs Computation* to find the pairs of accesses that are not protected by the same over-approximated set of locks held by the accesses. This is unsound, but is correct for their purpose of static race detection, if the objective is to not flag false positives. We adapted this approach to make it sound for our objective of eliminating instrumentation and included it as part of the base techniques when evaluating our work for race detection. We describe this adaptation in more detail in Chapter 7. In addition, we also use barrier sensitive analysis, by identifying the global synchronization points in the program. RacerX [36] is another static race detector that detects both races and deadlocks using flow sensitive inter-procedural analysis. Naik et al. [73] describe an algorithm to use conditional must not aliasing to improve the aliasing of pairs of objects in static race detection. Choi et al. [27] describes a reachability based program analysis method for Java programs to identify objects that can be allocated on stack, or are accessed by a single thread only. Their goal is to remove synchronization operation for those objects to improve performance. We do not use this technique to change the allocation location, though such techniques can be combined with our methods. Aldrich et al. [5] evaluate a set of methods to reduce synchronization overhead in Java programs. It uses static analysis to automatically remove unnecessary synchronization operations. Fang et al. [37] and Kuperstein et al. [59] describe memory fence insertion and optimization algorithms to guarantee sequential consistency. While the purpose

of their techniques is different, there are similarities with our methods to detect thread local accesses and identifying memory accesses that are race free.

Locksmith [81] is another static race detector that analyzes locks, and uses fork-join and mod-ref of memory locations to avoid instrumenting in single threaded code sections, and also includes some field sensitivity. However, SBPA further refines the contexts to global barriers, and augments the method with few directives that are highly effective. As SBPA works on LLVM IR, it is also language agnostic to some degree.

Agarwal et al. [4] introduce *may happen in parallel* analysis for X10, which is applicable to languages that adopt concepts of places, async, finish, and atomic sections from X10. Although there are some similarities, their analysis is path sensitive while SBPA is path insensitive. SBPA also targets a different programming model.

Voung et al. [97] proposed RELAY, a scalable tool to detect static races for millions of lines of C code under some optimistic assumptions. It modularizes races by limiting alias analysis per file, and does not handle pointer arithmetic. Therefore, RELAY has some sources of unsoundness. JLint2 [8] is another static program analysis tool to find synchronization issues in large scale Java programs.

Some researchers have also proposed to improve precision of static race detection using type systems. Flanagan et al. [39] and Abadi et al. [1] describe type based race detection for Java, that also considers thread local classes and common synchronization patterns in Java. Flanagan et al. [45] also proposed Houdini, a static annotation checker for modular Java programs.

2.2 Dynamic Race Detection

There is also a vast body of literature and tools covering dynamic race detection. One of the seminal works in this field was published by Lamport [61] in 1978, that described a simple model to represent the causal relationships and developed the notion of *happens before* among pairs of events. Most dynamic race detectors are based on this notion, and check that conflicting memory accesses are separated by a synchronization event. Savage et al. [88] proposed Eraser, a tool that detects data races dynamically in lock-based programs, and present several case studies demonstrating its effectiveness. FastTrack [41] is an optimized version of a precise dynamic race detector using vector clocks to track the *happens before* relationships. Ponzniansky and Schuster [79] developed a dynamic data race detector for C++ programs called Djit+. In theory, SBPA techniques could be integrated with any of these dynamic race detectors to reduce instrumentation overhead.

A very closely related work is Choi et al. [28], which uses static analysis to identify potentially racy accesses and instruments these accesses for use with a dynamic race detector. An earlier paper, Choi et al. [27], describes a reachability based program analysis technique to identify objects that can be stack allocated or that are accessed by a single thread only, where unnecessary synchronization can be removed. In both cases, the techniques presented targeted Java applications and, in contrast to our work, do not consider barrier or join operations during the static analysis phase.

Callahan et al. [75] proposed a hybrid dynamic race detector, that uses both lockset based detection and happens-before edges to improve the accuracy of dynamic race detection

and eliminate more false positives, with the main target application being web programs. Their optimization is dynamic in nature, not static as in SBPA.

Marino et al. [70] proposed LiteRace, a statistical sampling based data race detector that is light weight in runtime and memory overhead. They show that in catches more than 70% of data races by sampling only 2% percent of memory accesses. As their results show, this can miss some data races. And hence a static and conservatively correct optimization pass, such as SBPA, remains important.

RaceTrack [101] is an adaptive approach to detect races by focusing at the suspicious areas of the program to reduce overhead, that also requires a post-processing of the races to issue correct warnings. It is reported to have 2-3X slowdown at 1.2x memory overhead.

Zhou et al. [103] proposed hardware implementation of the lockset based race detection algorithm that runs efficiently. Their implementation detected 54 out of 60 injected bugs in the SPLASH-2 benchmarks. They concluded that a full hardware implementation requires significantly more hardware resources.

Bond et al. [19] developed a framework for detecting cross-thread dependencies and used for runtime analysis such as record and replay. Praun et al. [95] described an on-the-fly mechanism to detect races at the object level for Java programs.

Raman et al. [83, 84] developed a dynamic data race detector for use with structured parallelism in languages such as X10. They implement some of the same optimization as SBPA, such as eliminating checks in sequential regions of code and checks for read-only data in parallel code sections. However, the algorithms differ from those used in SBPA because their work is intended for structured parallelism.

Effinger-Dean et al. [35] proposed using *interference-free regions* to reduce the overhead of dynamic race detection. This work shares SBPA's goal of reducing instrumentation by segmenting an application, but takes a different approach in achieving this. Mellor-Crummey [72] developed one of the first dynamic race detection tools that also used static analysis to reduce instrumentation overhead. In that sense, it is similar to SBPA; but their tool was intended for Fortran, and the design and implementation differ from those of SBPA.

RedCard [44] describes eliminating redundant access tracking in dynamic race detection using static analysis. SBPA does not focus on redundant accesses; instead it eliminates conflict free accesses. Their method is orthogonal to what is proposed in this work. Cormac et al. also proposed Atomizer [40], a dynamic checker for detecting atomicity violations in programs. This is a stricter condition that race detection, since it checks for non-interference property of atomicity. Cormac et al. also proposed Velodrome [43], which is a sound and complete atomicity checker, that works on program traces and checks for conflict serializability. Unfortunately, Velodrome slows down program execution by an order of magnitude, similar to most race detectors.

Serebryany et al. [89] described ThreadSanitizer, a commercially deployed dynamic race detector developed at Google Inc. We combined SBPA with ThreadSanitizer, and present our results in this work. Another race detector in practice is Helgrind, which is part of Valgrind tool chain [90]. Valgrind also provides a dynamic analysis framework. In contrast, SBPA analysis is static, and can complement Valgrind to reduce race detection overhead.

2.3 Deterministic Runtime Systems

Recently, a number of works have focused on making multi-threaded programs deterministic, such that there is repeatability of results when running the same program with exactly the same inputs, assuming no asynchronous event occurs. While some languages such as StreamIt [93], Jade [87] and Parallel Haskell [25] are deterministic by nature, most compute intensive programming still use the C++ concurrency model [18] that do not use these specific languages.

SBPA was also originally developed for use in a deterministic runtime system using an always-on STM. Olszewski et al. proposed Kendo [76], a system that achieves weak determinism through deterministic locking among multiple threads. However, it doesn't achieve full ($strong$) determinism, which our system can enable by instrumenting the memory accesses. Our methods can be combined with Kendo and other such systems to achieve greater efficiency in full determinism. Edwards et al. [34] proposed SHIM, a model and language combination for deterministic software and hardware thread communication for embedded computing. Devietti et al. [32] proposed DMP, that enforces deterministic commit order for deterministic quanta of instructions. CoreDet [11] is a compiler and runtime environment that together enforces similar determinism. But it reports scalability issues, and its results are also sensitive to changes in compiler options or unrelated, minor changes in code. DThreads [68] describes a deterministic execution system, where threads are implemented as processes and OS memory protection enforces isolation between threads. DThreads does not rely on memory access tracking. It also avoids instrumenting single threaded code, but only in its runtime library through dynamic

checks, and not through static checks as we do. But it is unable to provide any insight into the causes of non-determinism, as it doesn't have a static compiler pass like SBPA. SBPA can not only make the program deterministic, but also help the programmer identify and fix the underlying issues.

2.4 Software Transactional Memory (STM)

Software Transactional Memory systems provide concurrency control for shared memory multi-processing via transactions. This can be viewed as an alternative to synchronization based on locking. Each transaction is a series of memory reads and writes, that are committed together if there is no conflict with other transactions. The successful transactions appear to occur at once to other processes.

The original idea of transactional memory was started as a hardware mechanism by Tom Knight [58] and was further popularized by Herlihy et al. [52], who described lock free data structures for shared memory processing and wait free synchronization [50]. Due to the cost of hardware at that time, the hardware version of transactional memory never became practical. However, in recent times, due to drastic reductions in cost and geometry of semiconductor devices, some limited support for hardware transactional memory has started appearing in the mainstream processor architectures [47, 71]. Due to the limited size of such support, STM still remains the only viable option for managing such transactions.

Shavit et al. [91] extended the idea to software transactional memory. Many STMs have been proposed and implemented since then. Herlihy et al. [51] proposed DSTM, the first

STM for dynamic sized memory objects. It maintains per object information in the form of locator objects to reduce conflicts at runtime. Harris et al. [49] described a new concurrency model based on transactional memory to compose larger abstractions of data structures. Harris et al. also proposed many different implementations of STMs based on the methods described in the paper [48]. Transactional Locking II (TL2) by Dice et al. [33] is an STM system based on commit-time locking and global versioning. RSTM [69] is a low overhead, obstruction free STM for non-garbage collected languages, RSTM is similar to DSTM in the design, as it is also a dynamic STM. TinySTM [38], a word based STM based on locking, is one of the best performing STMs with lower single threading overhead.

Most of the STMs listed above focus on reducing the overhead at runtime with various techniques. Our work focuses on the static reduction of such overhead. There has also been prior work to reduce STM overhead, mainly for strongly typed languages. Beckman et al. [10] described object ownership in Java to reduce logging operations in STM. Blanchet [16] described escape analysis for object oriented languages like Java. Li and Verbugge [67] describe similar work on *May Happen In Parallel* for Java, that performs flow sensitive analysis of locks and synchronizations, which is runtime intensive. Our analysis does not rely on locks, and uses a different approach of program segmentation orthogonal to theirs. SBPA can complement ad hoc synchronization detection mechanisms, such as Xiong et al. [100], Tian et al. [94] and Jannesari et al. [54]; but currently does not use such methods and relies on global barriers only. Afek et al. [3] describe static analysis techniques to reduce STM overhead that are similar to the ones implemented in CoreDet [11] and used in our baseline system. While their method focuses on some optimization specific to code motion and read-only transactions, we developed

a framework for identifying different program sections to harness different data usage in those sections in reducing instrumentation.

Cascaval et al. [24] argue that STM overheads are so high, that STM is merely a research toy. With our proposed techniques, we believe STMs can become significantly more efficient, and thus also practical.

2.5 Data Flow Analysis

Data flow analysis is used in multiple areas of compilers, such as reachability analysis, liveness analysis, def-use analysis etc. Program control flow is also an integral part of data flow analysis. Kildall [57] formalized the data flow analysis problem by representing the analysis as a set of equations at different basic blocks. Sharir and Pnueli [77] described two possible approaches to the inter-procedural data flow analysis. They called their first approach a `functional` approach that considers functions as super operations, defined by the effects the function calls produce on global data. Their second approach, `call strings` approach, blended the intra-procedural and inter-procedural approaches together by creating a global flow graph. Most global data flow analysis problems are solved by iterative methods [22, 56, 78], where the iterations stop once a fixed point is reached. Pointer analysis, a form of data flow analysis to infer points-to property is used extensively in SBPA, and so we discuss it separately in the next section.

2.6 Pointer Analysis Background

In this section, we briefly discuss some background on pointer analysis, which is a form of data flow analysis. Pointer analysis or points-to analysis, is also sometimes referred to as *alias analysis* or *shape analysis*. It tries to find non-trivial relationships among pointer variables and memory locations. It is related to alias analysis, which answers the question of whether two pointer variables p_1 and p_2 point to the same memory location. An alias analysis can answer the question in three ways: may alias, must alias or do not alias. The alias information can be very useful for several compiler optimizations and program understanding analysis, such as live variable analysis, register allocation, constant propagation and compile time checking for potential runtime violations. Raman [85] provides a brief survey of various pointer analysis techniques.

A trivial points-to set for a type-unsafe language is where each pointer points to every memory location, and hence each pair of pointers may alias each other. If a language is type-safe, meaning pointer variables of different types cannot be assigned to one another, then a slightly better trivial points-to set can be achieved, where each pointer only points to all memory locations assigned to that particular type, and alias with only the same type of pointers. Thus, any pointer analysis approach has to improve upon these trivial solutions by finding more precise points-to sets.

A precise pointer analysis is very important for many program analyses. Hence, the subject of pointer analysis has been researched extensively for more than three decades. In their 2001 paper, Hind et al. [53] provide a description of the still open issues in this field. Many

types of algorithms have been proposed for the problem. The approaches range from very precise but exponentially expensive flow sensitive, path sensitive analyses to flow insensitive, path insensitive algorithms which are generally very efficient in runtime and memory overheads. The analysis can also be distinguished by whether they consider whole program (inter-procedural) or limit the analysis to one procedure at a time (intra-procedural). Below, we describe some of the most prevalent methods. The reader can refer to the references in this section for more in-depth understanding.

2.6.1 Flow sensitive vs. insensitive pointer analysis

A flow sensitive pointer analysis considers the control flow of the program to create a points-to set for each variable at different program points, either at basic block level or at finer granularity of instruction level. For example, consider the code excerpt in Example 1. In this example, the variable x might point to either a or b depending on the location of the code. A flow sensitive analysis will thus have two different points-to set for x, whereas a flow-insensitive analysis will just have one points-to set for both basic blocks for the code fragment, which is the union $\{a, b\}$.

Flow sensitive pointer analysis is challenging for large programs, as the time and space complexities can become exponential. Hardekopf et al. [] proposed a staged flow sensitive analysis for millions of line of code, that balances precision with complexity with equivalence classes for def-use pairs. Choi et al. [26] also proposed practical approximation methods for flow sensitive pointer analysis using compact representations, where they trade off some precision.

Example 1 An example of flow sensitive points-to set

```
void func(int *a, int *b)
{
  int *x;
  if(a < b) {
    x = a;
    ...
  }
  else {
    x = b;
    ..
  }
}
```

2.6.2 Context sensitive vs. insensitive pointer analysis

A context sensitive analysis will have a different points-to set for the same pointer in different program contexts. The context can be chosen in many different ways, depending on the requirements of the analysis. For example, the context could be different call sites for a function as shown in Example 2. A call site of a function F is an instruction in the program that calls function F. A context sensitive analysis would re-analyze the points-to sets for function variables for each call site, merging the function arguments at the call site. Thus, the variables within the function can potentially have multiple points-to set depending on the call sites from where the function is called. In Example 2, the function $func$ of Example 1 is called from two different places. For call site 1, x points to $\{A, B\}$, whereas for call site 2, x points to $\{C, D\}$. But a context-insensitive analysis will have a single points-to set for x, which is $\{A, B, C, D\}$.

Inherently, flow sensitive and context sensitive analyses are exponential in complexity due to the exponential number of possible recursive contexts and execution paths in the program. Whaley et al. [98] propose a cloning based context sensitive pointer analysis, in which they

Example 2 Function func() called at multiple call sites

```
int A,B,C,D;
...
func(&A, &B);              // 1st callsite
...
func(&C, &D);              // 2nd callsite
```

clone functions in each context of interest for the acyclic call paths. Berndl et al. [13] proposed a new way of solving the subset based pointer analysis using binary decision diagrams [20] (BDDs). BDDs were originally proposed by Bryant for their application in equivalence and model checking for electronic circuits, but they can also be used for compact representation of sets for pointer analysis. Burke et al. [21] proposed a flow insensitive inter-procedural alias analysis, that preserves some of the precision lost in a flow insensitive technique.

2.6.3 Dynamic vs. Static pointer analysis

The previous methods are static analyses performed during compilation, and at best they can achieve the accuracy of the worst case dynamic scenario. Dynamic analysis tries to collect points-to information during runtime for different variables. This can be used to compare the precision of static algorithms. In case of probabilistic pointer analysis, it can predict the probability of a pointer pointing to certain memory locations. It can also yield interesting profiling based optimizations. Axel [9] presents some interesting comparison of static vs. dynamic pointer analyses, and shows that the static sets are quite often too big and inexact for use in program understanding. The dynamic sets found are much smaller and in 97% of the cases actually singletons with averages close to 1. Dynamic points-to data can

enhance program understanding and compile-time optimizations greatly. Some of the uses are more exact alias analysis, feedback driven compilation and program understanding methods like program slicing. But dynamic methods depend on the input sets used to train the points-to analyses, and so they are generally more difficult to apply.

2.7 Some important Flow insensitive methods

Due to the complexity of flow sensitive, fully context sensitive or dynamic pointer analyses, most practical analysis methods are static, flow insensitive. These completely ignore the control flow graph, and compute only one points-to set for a variable for the entire program, or the scope of analysis. The order of instruction execution is ignored completely. A few notable flow-sensitive algorithms are described below.

2.7.1 Andersen's flow insensitive analysis

Andersen [7] proposed a method in which each pointer variable assignment is treated as a constraint. Then, the analysis would propagate these initial constraints transitively till a fixed point is reached. The solution to the pointer analysis problem is the fixed point solution of the transitive closure of the initial points-to graph. Example 3 shows a small code segment of how this works. Since pointer a points to memory location A in line 1, $a \supset A$. Similarly, due to line 2, $b \supset B$. The assignment operation $b = a$ in line 3 causes an additional constraint (edge), $b \supseteq a$. The rest of the initial constraints shown in column 2 are derived similarly. The initial constraints define the initial points-to relationship graph. Final constraints are found by

an iterative algorithm to find the transitive closure of the initial directed graph. Since the best known algorithm for transitive closure is $O(n^3)$, the worst case complexity of this analysis is also $O(n^3)$, where n is the number of pointer variables in the program.

Example 3 A code excerpt to explain Andersen's points-to analysis

Code	Initial Points-to	Final Points-to
1: a = &A;	a -> {A}	a -> {A}
2: b = &B;	b -> {B}	
3: b = a;	b -> {a}	b -> {A,B}
4: c = &C;	c -> {C}	
5: d = &D;	d -> {D}	d -> {D}
6: c = d;	c -> {d}	c -> {C,D}

2.7.2 Steensgaard's algorithm

Steensgaard [92] described an almost linear time flow insensitive points-to analysis for a strongly typed language. It uses type inferencing to compute a non-trivial flow insensitive, inter procedural points-to graph. Steensgaard's analysis can be viewed as a simplification of the previously described Andersen's algorithm, in which an assignment of pointer variables results in unification of both LHS and RHS sets.

Thus, whereas Andersen's algorithm would find a transitive closure of the points-to graph, Steensgaard's algorithm would only do repeated union-find operations for pointer assignments. The sets of pointer variables and memory addresses are represented as inverted trees. At every union operation, the paths to the roots of the trees are compressed, thereby reducing subsequent union costs. It also uses union by rank, meaning the tree with lower depth is merged into the root of the tree with higher depth. The runtime complexity of the algo-

rithm is $N \times Ack^{-1}(N)$ where Ack is the Ackermann function. This is almost linear, since $Ack^{-1}(N) < 4$ for $N <= 2^{132}$.

Example 4 shows the computed Steensgaard points-to sets for the same code as in Example 3.

Example 4 A code excerpt demonstrating Steensgaard's points-to analysis

```
    Code        Initial Points-to  Final Points-to
 1: a = &A;       a -> {A}
 2: b = &B;       b -> {B}
 3: b = a;        b -> {a}                {a,b} -> {A,B}
 4: c = &C;       c -> {C}
 5: d = &D;       d -> {D}
 6: c = d;        c -> {d}                {c,d} -> {C,D}
```

Chapter 3

Finding Disjoint Thread Sections

To find the single threaded, multi-threaded and disjoint thread sections in a program, we use techniques from program flow analysis. Our overall methodology involves building a reduced inter-procedural control flow graph (RICFG) of the program, and then performing control flow analysis on the RICFG to identify the aforementioned sections of the program. Thereafter, the section information is used to improve the instrumentation flow for race detection, deterministic execution and other use cases. In the next section, we review some compiler terminologies and other terms used in this chapter.

3.1 Terminologies

We described some common terms used in the rest of the text. A more in-depth description of these terms can be found in the book by Lam et. al. [60].

A *basic block* is a sequential block of instructions without any branching or termina-

tion statement within it, or any incoming edge into it. All execution of the block starts at the first instruction and exits at the last instruction.

The *control flow graph (CFG)* of a program shows the flow of execution for the program, where each node is a basic block and each edge represents a transfer of execution from one basic block to another. The program start is the source node S in the flow graph. All exits from the program are merged at the sink node T of the flow graph. CFG can also be constructed for a segment of the whole program, such as a function.

Prosser introduced the term *dominance* in his 1959 paper [82]. In the control flow graph, the *dominator* of a basic block B_i, denoted Dom(B_i), is the basic block closest to B_i that must appear in any execution path that reaches B_i from program start node S. Thus, every node dominates itself by this definition.

A *dominator tree* is another representation of dominators, in which each node only dominates its descendants. And thus, there is a unique dominator for each node in this tree. For two basic blocks B_i and B_j, the dominator is the lowest ancestor of Dom(B_i) and Dom(B_j), both inclusive, that dominates both Dom(B_i) and Dom(B_j). The notion can be extended to a set of basic blocks $B_1, B_2, \ldots B_n$, where $Dom(B_1, B_2, \ldots B_n)$ is the lowest ancestor in the CFG that dominates the dominators of basic blocks $B_1, B_2, \ldots, B_n$.

Similarly, the *post dominator* of a set of basic blocks $B_1, B_2, \ldots B_n$ post dominates these blocks; i.e. it is always executed after any of these blocks are executed.

A *loop* L in the CFG is a portion of the CFG, in which there is one entry point called *header* that dominates all the other nodes in L, and there is at least one back edge pointing back to the header. Sometimes, compilers introduce a pre-header block right before the header,

and a post-header block right after the loop, so code within the loop that doesn't depend on the variables modified in the loop can be moved into these outer blocks for reducing redundant executions.

A *callsite* for function F is an instruction that transfers program control to function F. In other words, call sites of a function are the places from where a function is invoked.

Pointer Analysis is a form of data flow analysis that identifies the aliasing information among different pointers through static or other methods. See section 2.6 for a detailed background on this subject.

We define a *code section*, represented by a tuple (B_m, B_n) where B_m and B_n are basic blocks in the CFG, as a set of all basic blocks that have B_m as their common dominator and B_n as their common post-dominator. Note that the code section includes both B_m and B_n.

A *thread section* is a collection of code sections that *might* be executed concurrently.

Figure 1.1 shows an example execution, with two thread sections TS1 and TS2, and four different code sections (CS1, CS2, CS3, CS4). TS1 consists of CS1 and CS2, whereas TS2 consists of CS3 and CS4. Although not shown in the example, same code section can appear in more than one thread section when multiple thread sections execute the same code.

We define two types of thread sections as follows. Later, we explain how these are identified by our analysis.

Single-threaded thread section *Single-TS* (Section 3.4) is a collection of code sections that can only execute when a single thread is active.

Disjoint thread section *Disjoint-TS* (Section 3.5) is a collection of code sections that *might* be executed by multiple threads simultaneously. Note that same code sections can ap-

pear in multiple Disjoint-TS. A union of all the Disjoint-TS sections constitutes a conservative closure of multi-threaded code regions of the program.

Praun et al. [96] use the term *conflicting* for potentially unsynchronized accesses to shared objects. We define *conflicting* read-write (or equivalently load-store) as a pair of read and write of the same memory location by different threads that are executing concurrently, without a happens-before edge between them. For data-race detection, conflicting loads and stores must be synchronized with barriers and/or mutual exclusion checks. For deterministic execution, the requirement is stricter, since the lock acquire-release orders can affect the program output. So, even the lock protected accesses are treated as conflicting for our analysis. Thus, a conflicting access can be racy or non-racy. We define the other accesses separated by happens before edges as *non-conflicting*. A non-conflicting memory access by a thread does not affect the behavior of other threads executing concurrently.

In this work, a *memory location* implies an abstract set of memory addresses that are seen as equivalent by pointer analysis used in SBPA. We use read/load for a memory read access, and write/store for a memory write access interchangeably.

3.2 SBPA Pointer Analysis Framework

SBPA uses the DSA pointer analysis framework [65] [63] which is a unification based [92] pointer analysis framework. This is also used in CoreDet [11] from which we derived our baseline set of optimizations (henceforth referred to as *Base*) used in our evaluation. In addition, DSA works with the LLVM compiler framework [64] which is used by SBPA. DSA

is context sensitive in the sense that it can be applied to a local context of the program. However, both SBPA and *Base* need to identify the potential points-to set for the whole program to detect where a pointer used in an instruction may point, and whether it conflicts with any other access in the whole program. Therefore, for SBPA the analysis is performed in the global context. The pointer analysis first creates a local points-to graph for each function in a bottom-up manner. For a function call, the points-to sets of formal function arguments are merged with those of the actual arguments. We modified DSA as described below for improved results. Note that some of these techniques are covered in literature in various forms [102] [28] [62]. We include these here to provide a complete explanation of the differences with $Base$ method, The use of these techniques is categorized as *Alias* in our results section.

3.2.1 Perform modref analysis per section

We performed modified and referenced analysis for the points-to sets (memory regions) for each Disjoint-TS. For each points-to set, two separate bit arrays R and W are maintained for read and write accesses respectively. If $R[i]$ is true for a points-to set S, it implies some instruction in Disjoint-TS i reads some memory location in S. Similarly, $W[i]$ indicates a write to any location in S in Disjoint-TS i. A depth-first traversal for each section i sets $R[i]$ and $W[i]$ of the accessed points-to sets.

3.2.2 Adaptive non-unification for points-to sets of function arguments

Generally, in unification methods, two points-to nodes (abstract memory locations) are merged when any pointer can point to both of those nodes. In DSA [65], the calls to a

function f are resolved by merging the points-to nodes of actuals with those of the formals of f at each callsite. This ultimately leads to the pointers corresponding to arguments of f pointing to an unified node that includes all the actuals at different call sites. So, if $f(X)$ and $f(Y)$ are two such calls, and $P(x)$ represents the points-to node of x, then this leads to a merger of $P(X)$ and $P(Y)$. However, if X and Y are accessed in a section where $P(Y)$ has a write and $P(X)$ only has a read, this false aliasing caused by merging $P(X)$ and $P(Y)$ results in instrumenting read-only accesses of X in that section.

To circumvent this issue, we modified DSA to avoid these mergers of actuals for some functions. To compute the points-to node for a pointer within f in the program context, we must then recursively traverse the function call graph in a bottom-up manner until the actual is not a function argument passed from above. This can lead to deep recursions, and an exponentially growing number of points-to nodes for the pointer. So, we used three thresholds to constrain the runtime and memory requirements of this scheme. First, this was only used for functions with at most $T1$ callsites. Second, if the pointer argument of f was passed many levels through the function call graph, then f was resolved by merging its arguments with actuals. This can be detected during the upward traversal in the call graph starting from each callsite of f. If the depth of such argument passing exceeded $T2$, then calls to f were also resolved by the original method. Third, if during the process, the number of elements in a points-to set of any argument of f exceeded another threshold $T3$, then the most commonly occurring elements within the set were merged to reduce the size of points-to set to $T3$. For its implementation, these non-unified points-to sets were maintained as a `map`, with each points-to node mapping to its occurrence count within the set. Whenever the size of the set exceeded $T3$, the two nodes

with the most occurrences counts were merged into a single points-to node, making them indistinguishable. The calls to functions in strongly connected components in the call graph were resolved by the original method of unification.

For our experiments, we used $T1$ = 4, $T2$ = 4 and $T3$ = 10. To resolve the points-to set for any formal argument of a function, it can take at most $T1^{T2}$ recursions. Since both $T1$ and $T2$ are small constants, the worst case runtime of such computation is $O(1)$. $T1$ and $T2$ should not be large, as that leads to a high constant multiplier. The third threshold guarantees that the size of non-unified sets are within $T3$. Once the above process is completed in a top-down manner, the non-unified points-to sets of all function arguments that were not merged are available for use by SBPA analysis, and recomputation is not required. The benefit derived is the disambiguation of points-to sets that would otherwise have been merged if they appeared as actual arguments in multiple calls of the same function. Very small functions are inlined by the compiler, so their formals aren't merged by unification in most cases, thus making the pointer resolution within these inlined functions context sensitive. So, our above heuristic only affects function call resolutions for relatively large functions.

The result is never worse than the original approach, since the different actuals would always have been merged otherwise. The pointer analysis remains context insensitive; only the mod-ref analysis becomes context sensitive for the points-to nodes not merged by this approach. We found this hybrid, adaptive approach to be less memory and runtime intensive while providing significant disambiguation benefits by making the mod-ref analysis more precise. As described above, we performed this non-unification strategy only for function calls. At other places, as in function bodies, the points-to nodes are unified as usual.

3.2.3 Field sensitivity for array elements

DSA is field sensitive only for small sizes of pointed memory objects. For large arrays, it loses field sensitivity as it collapses the points-to node to a size of 1. We modified it to reduce it to the size of the elements in the array. Thus, the array indices alias for all accesses, but the offsets within the array element still remain distinct. We added a check for strided accesses to unalias the store to one field from load from another field. To minimize performance overheads, we used a simple scheme based on offsets from the beginning of a structure's memory. Assume a read r and a write w in the same Disjoint-TS accessed data through the same points-to set that has elements of type *struct ss*. Also assume that the access sizes are s_r and s_w respectively, at offsets of o_r and o_w respectively within the struct. In such a case, if $o_r + s_r < o_w$, or $o_w + s_w < o_r$, there is no overlap, and so r and w can be classified as non-conflicting. This helped significantly for arrays of structures, where in some phases, only a certain field is written while other fields are only read.

3.3 Constructing the Reduced ICFG

SBPA requires an inter-procedural control flow graph (ICFG) of the whole program to understand the multi-threaded structure of the program. Since it is used only to detect the multi-threaded regions of a program, and is an auxiliary graph that points back to original CFGs of functions, some optimizations can be performed to reduce the size of the ICFG. We call this reduced ICFG (*RICFG*) in subsequent discussions.

Algorithm 1 is a high level description of creation of the RICFG. The RICFG of the

top level function `main` is constructed recursively. In the RICFG, if a function f or any function called within f has any thread related directive, we replace the call to function f at all its call sites with the RICFG of f. To do this, first the RICFG of f is created in a recursive manner. In RICFG of f, a common return node $R(f)$ is created that has incoming edges (R_i, $R(f)$) for each of the original return nodes R_i in RICFG of f. Then, for each call site of f, we create two nodes $N1$ and $N2$ for the basic block containing the call instruction. The call to f is replaced with an edge from $N1$ to the entry node in RICFG of f, and another edge from $R(f)$ to $N2$. The nodes in RICFG of f point back to original CFGs of functions, which remain unmodified.

Algorithm 1 Constructing RICFG for a function F.

```
1.start = First instruction of F
2.Copy CFG of F to its RICFG; with RICFG
  nodes pointing to original basic blocks.
3 foreach basic block B in F:
 .1 foreach instruction I in B:
  .1 if I is a call to function f with spawn, join or sync,
       or if I is a call to spawn a thread:
    .1 ricfg(f) = Create or find RICFG for callee function f at I.
    .2 Create node B1 for instructions in B before I.
    .3 Create node B2 for instructions in B after I.
    .4 Add edge from B1 to entry of ricfg(f)
    .5 Add edge from return node R(f) of ricfg(f) to B2.
  .2 else if I is a global barrier call:
    .1 n = find or create global sync node for used ID
    .2 Create B1 and B2 as above, partitioning B at I
    .3 Add edges B1 to n, and n to B2.
4. Remove exit edges from RICFG.
5. Build the dominator and post dominator tree for the RICFG.
```

Note that we do not insert the whole CFG of f at its call sites, but only add two edges. Thus, there is no scalability issue for this approach for large programs. For each call instruction,

at most 2 edges are added to the RICFG, and at most two nodes are created for the basic block containing the call site. For our purposes of identifying threaded sections, we are only interested in analyzing the control flow of functions that contain thread spawn, join, barrier or other thread directives. If some function does not have these directives, we do not need the inter-procedural flow of control through that function. This eliminates the need to expand a vast majority of the basic functions. So, all library functions and external functions do not need to be expanded in this RICFG. This is why we call it *reduced* ICFG. Algorithm 1 gives a high level view of creating the RICFG. As seen in the algorithm, it is a recursive process. A function is considered to have one of the interesting thread directives, if either the function or any descendant function called from within that function has these directives; so this is a recursive relationship. The directives of interest here are thread creation, join, cancellation and *sync_barrier*.

The functions with interesting thread related directives are marked in a global hash table, so each function is inspected only once on whether it has a thread directive. Similarly, when an ICFG of a function is created, its entry node and the return nodes are recorded in a hash table organized by this function. So, subsequent queries to the function simply return information for the already constructed RICFG of the function, which is a part of the global ICFG for the whole program. Then, edges are added from/to those nodes as described earlier, to complete the ICFG construction of the caller function. Since we add one edge from preceding instruction to the entry node of called function, and add one edge from common return node to the successor instruction, we insert at most 2 edges for a call instruction. Also, each basic block can be split at at most all the call instructions it contains, and at the $sync\_thread$ calls it has. Therefore, the ICFG has at most N nodes, where N is the number of instructions in the

program. Also, for each function, it adds one common return node for all return statements. If there were M return statements, it leads to M edges for all the functions. Also, if total number of callsites is C, then we add an additional $2C$ edges to connect the ICFG of a functions to its callsite. The runtime is proportional to the number of instructions in the program, as each instruction is visited at most once in the BFS traversal shown in 1. In reality, our approach expands only a few high level functions and so space and runtime overhead is not a concern for even large programs.

We explain the creation of the RICFG with the example in Figure 3.1. It shows a part of a basic block B within a function F. There is a call to function f, and f has one of the thread directives that we are interested in. In the process, an RICFG is first created for f. The basic block B is then represented by two nodes $N1$ and $N2$, with $N1$ comprised of the first part of B and $N2$ comprised of the latter part of B. $N1$ then points to the node for entry of function f. $R(f)$ is the consolidated return node for all returns within f. In other words, the incoming edges in $R(f)$ are from the original return statements in f. An edge is added from $R(f)$ to $N2$ to maintain the control flow in B. This is a recursive process. So, the RICFG of f is created if it isn't created already when its call is encountered first. Also, the splitting occurs repeatedly. Thus, if the instructions in $N2$ contain any call to some other function f' or f itself, $N2$ will be split again at that call. If the called function doesn't have any of the thread directives that defines the sections, then the splitting does not occur.

Once the RICFG is constructed, we splice the exit edges to simplify post-dominance. The exit edges due to assertions or exit() are not useful for SBPA analysis, and also break the post-dominance relationships that exist in normal program flow. Maintaining the post-

dominators are important for SBPA to find the single entry single exit [55] nodes in the RICFG.

Once we have removed the exit edges and created the RICFG, we construct the dominator and post dominator tree for it. There are many algorithms in literature [6, 29, 30, 66] to build the dominator tree for directed graphs, such as CFGs. Lengauer et al. [66] proposed an almost linear time algorithm. Georgiadis and Tarjan [46] improved the algorithm to be linear time. However, for simplicity of implementation, we used [29], which is an iterative algorithm to build the dominance frontier. Even though its worst case asymptotic complexity can be quadratic in number of nodes, it is faster in practice as the worst case is rarely encountered.

Note that the nodes in RICFG point back to the original basic blocks, which are part of the original, unmodified CFGs for the containing functions. Thus, it is an auxiliary data structure besides the original CFGs of each function. The nodes in RICFG contain the following information:

- Original basic block it represents
- Start position in basic block
- End position in basic block
- Unique ID of *sync_barrier* in case this represents a *sync_barrier* node
- A list of successor nodes in RICFG from this node
- Number of incoming edges. Note that we don't need the back edges, as the SBPA algorithms never traverse backwards.

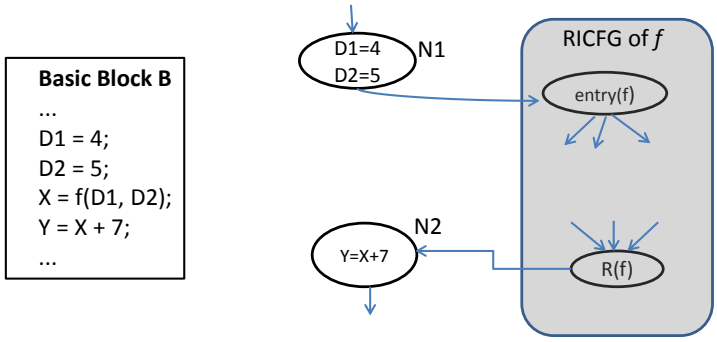

Figure 3.1: A code snippet showing the call of a function f, that has any of the thread creation, join or synchronization directive. The resulting changes in RICFG are shown on the right.

3.4 Single-Threaded Thread Sections (Single-TS)

Intuitively, multithreaded applications have thread sections because the programmer is responsible for ensuring that inter-thread communication is properly synchronized. The synchronization directives the programmer uses naturally divide the program into thread sections, and the communication pattern in each thread section can be detected at compile time.

Programming Patterns: As defined earlier, Single-TS code can only be executed in single-threaded mode. Single-TS exist because some code is safe to be executed only when a single thread is active. All the concurrent threads are either not created, or not executing in such mode. Application initialization and finalization in many programs fall in this category. In many applications, parent thread waits for all the spawned threads to complete before running some finalization code. Also in the fork/join programming model it is common to alternate between parallel and serial regions. In all these cases, the serial regions could be detected as

Single-TS, thereby eliminating the need to detect unsynchronized communication or races in these code sections.

If a memory location is determined to be modified only in Single-TS, all reads to the same memory location in any thread section are race-free. Then, identifying Single-TS in a program can not only help eliminate instrumentation in the Single-TS regions of application, but also in the regions that are not Single-TS.

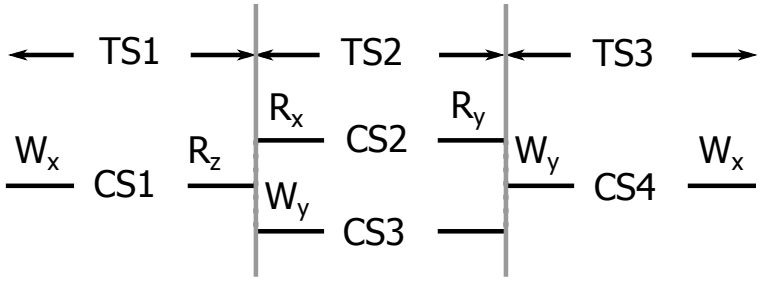

Figure 3.2: A program comprised of 4 code sections in 3 thread sections

Detailed Example: Figure 3.2 shows a program divided into 4 code sections appearing in 3 thread sections TS1, TS2 and TS3. CS1 and CS4 are in single-TS TS1 and TS3 respectively, while CS2 and CS3 can execute concurrently within TS2. CS1 and CS4 are single threaded, so they don't need instrumentation. Memory location X is only written in CS1 and CS4, and is accessed only for reads in TS2. So, no instrumentation is required for X in any of the thread sections. Only Y needs instrumentation in TS2, as it is both modified and read in the same Disjoint-TS.

Algorithmic Solution: Algorithm 2 is a high level description of our method to detect code sections executed in single-threaded mode only. The algorithm marks code sections as either single-threaded or multithreaded. If a code section can be run in both single-threaded and

multithreaded sections, it conservatively marks that as multithreaded code section (MTCS). While it is possible to replicate functions that are called in both Single-TS and MTCS, and assign one copy to Single-TS to avoid instrumentation in that, in the benchmarks we studied, we did not see opportunities for this, and hence this option was not explored.

Once the program is segmented into these sections, we perform an improved points-to analysis to extract modified-referenced information for memory locations accessed in overlapping thread sections (described in Section 3.2). These techniques, combined together, help identify the real conflicting loads and stores in the program. Loads from a memory location that happen in a thread section, which do not overlap with any other thread section that writes to the same location, are data-race free, and so do not need to be instrumented since these loads are *non-conflicting*. Similarly, if a store is data-race free, it does not need to be instrumented in tools that check data races or provide determinism. However, STMs still need to log or version the data in stores to support restarts that may be triggered by conflicts with other memory accesses in a transaction. So, even data-race free stores in Disjoint-TS might need to be instrumented in STMs. We call these *log-only* stores. Thus, SBPA classifies the stores as conflicting, non-conflicting and log-only.

The algorithm finds multithreaded code sections (MTCS). Single-TS code is any code that is not included in any MTCS. It first creates a Reduced Inter-Procedural Control Flow Graph (RICFG) as explained in 3.3. Thereafter, it splices away the exit edges, such as asserts and exits for error conditions. These are irrelevant for our analysis of which code can execute in parallel. The reason we splice out these exit edges is that it simplifies the RICFG for the purposes of post-dominance relationship. These two steps are shown as steps 1 and 2 in Algorithm 2.

Algorithm 2 Finding multi-threaded code sections (MTCS).

```
1 Create RICFG for top level function main
2 Remove exit branches from global RICFG
3 foreach spawn call c in the RICFG:
  .1 if a join j is control equivalent with c:
    .1 code between c and j is multithreaded
  .2 else if c is in a loop:
    .1 foreach control equivalent loop with same iterations:
      .1 if the loop join calls always match c loop:
        .1 code between c and j is multithreaded
  .3 else:
    .1 code between c and exit is multithreaded
```

The next step is to iterate over each spawn call (step 3). If the spawn call is followed by a single join call, and both are control equivalent [1] (step 3.1), all the code sections dominated by c and post-dominated by j are marked as MTCS (step 3.1.1). Since we use the RICFG for this, it includes all the function calls embedded in this section, and the functions invoked by the thread spawns.

A more common case is where a loop that spawns a number of threads is followed by a similar loop that joins or waits until all the threads have finished execution. This is a typical fork/join program pattern in parallel applications. Both spawn calls and join calls should be in control equivalent loops that have the same number of iterations (step 3.2.1). In addition, we check that the join and spawn calls inside the loop are not control dependent, which guarantees that both loops call exactly the same number of spawn and join calls (step 3.2.1.1). When these conditions are met, we mark the loops and the code sections dominated by the spawn loop and post-dominated by the join loop as MTCS.

[1] Two nodes (x and y) in a control flow graph are control equivalent if x dominates y and y post-dominates x.

When we cannot find the control equivalent join of the loop, we conservatively assume that the rest of the program is multithreaded (step 3.3.1). This ensures that the single-TS identification is conservatively correct.

3.5 Disjoint Thread Sections (Disjoint-TS)

Programming Patterns: Figure 3.3 shows an example of the regions in program that we want to identify with our analysis. The parent thread creates p child threads in this threaded section. Each of the child threads are further decomposed into smaller spans $T(1,1), T(1,2)$ and so on separated by global thread barriers. The entire thread section begins at basic block S and terminates at T, where S to T is a flow graph embedded in the program CFG. Since these sections are separated by global barriers, where all these threads must synchronize, we call these *disjoint thread sections (Disjoint-TS)*. For this example, we name them as DTS(1), DTS(2), ... DTS(n).

Since they never overlap in any execution trace, we can segregate memory accesses into disjoint sets of accesses corresponding to each of DTS(1), DTS(2), ... DTS(n). Hence, when we instrument memory accesses in each of these, we only need to consider the modified and referenced information within the context of the disjoint thread section in conjunction with the parent thread. The parent thread usually waits and synchronizes the child threads, and so we consider it to be overlapping with all thread sections in its entirety for all sectional analyses. However, different programming patterns do exist. And sometimes the parent thread also execute the same or a different function in parallel with the child threads. In such cases, the

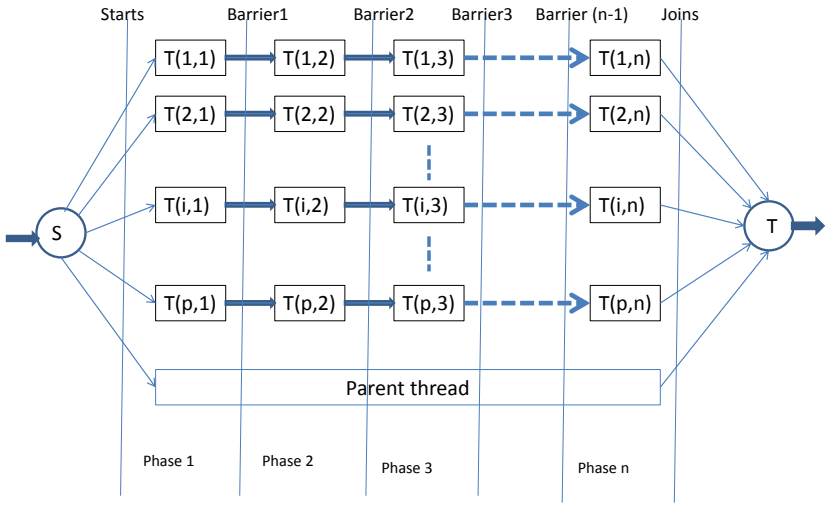

Figure 3.3: A decomposition of threaded section into smaller, disjoint parallel segments

parent thread must also execute the same barrier synchronizations as the children threads for such segmentation to work.

Code appearing in any Disjoint-TS can potentially run in multi-threaded mode. So, loads and stores in these sections need to be instrumented unless proven unnecessary. Portions of Disjoint-TS can overlap with each other. While some code sections of a Disjoint-TS is executed only by one Disjoint-TS, others might be executed by many Disjoint-TS.

In general, Disjoint-TS occur due to non-overlapping parallel phases separated by Single-TS code sections, or due to global synchronization barriers within same parallel phase. The finer-grained partitioning helps identify non-conflicting reads and writes within parallel phases with increased precision.

Detailed Example: Figure 3.4 shows a program with 5 different code sections, CS1 through

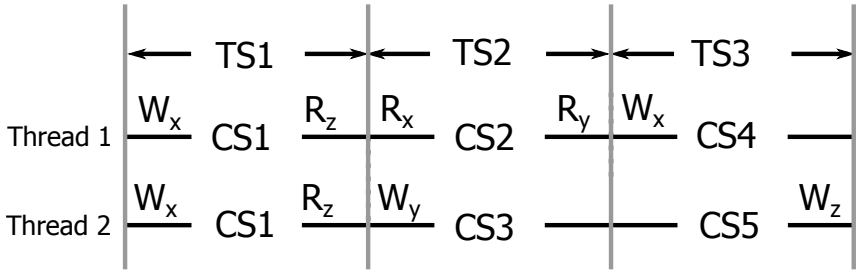

Figure 3.4: Two threads executing 5 code sections in 3 disjoint thread sections.

CS5, which read and write memory locations X, Y, and Z. Since all memory accesses are performed in multithreaded thread sections, without Disjoint-TS, all the accesses are assumed conflicting. However, the barriers guarantee that some accesses cannot overlap. Reads and writes to Z in code sections CS1 and CS5 belong to non-overlapping thread sections TS1 and TS3, so R_Z in TS1 is not racy. Similarly, R_X in TS2 is data-race free as there are no writes to X in TS2. Accesses to Y in CS2 and CS3 may be racy, and must be instrumented. Stores are instrumented, as we can't determine how many threads are executing the same code section within a thread section. If the memory locations are thread local, then for data race detection, the stores don't need instrumentation. For STMs, stores to local memory are race free. But SBPA cannot prove that there is no restart due to some other conflict. So, those stores for STMs are instrumented as log-only stores.

Algorithmic Solution:

One issue with *pthread_barrier_wait* function in POSIX library is that the programmer can initialize the barrier variable with incorrect value, leading to deadlock and potentially other unexpected behaviors. Example 5 shows a bad code snippet where barrier is initialized with $nthreads$, but $nthreads+1$ threads are created, each presumed to call *pthread_barrier_wait*

Example 5 Incorrect initialization of barrier value can lead to deadlock and incorrect global barriers.

```
// Initialize with smaller
pthread_barrier_init(&global_barrier, NULL, nthreads);

for(int i = 0; i <= nthreads; i++)
  pthread_create(&threads[i], NULL , simulate, &args[i]);
  ...
```

to synchronize among themselves. In the simple example shown here, it is possible to identify the issue statically. However, in more complex cases, we cannot determine if the barrier is initialized properly.

Algorithm 3 Finding disjoint thread sections in MTCS

```
1 create RICFG for top level function main
2 foreach thread section (ts) found by Single-TS:
 .1 if the ts is not single threaded:
  .1 let b = beginning of ts
  .2 let e = end of ts
  .3 while b != e:
   .1 C = reachable barrier nodes starting from b.
   .2 if C has a single node:
    .1 Code between b and C is a new thread section
    .2 b = C
   .3 else exit search:
```

Other factors, such as code duplication by compilers for some other optimizations, different threads executing different functions, barriers across multiple functions and barriers within loops complicate the identification of these global barriers where all the threads synchronize. Not all the threads may reach the same pthread_barrier_wait() instruction to synchronize. Hence, we propose a *sync_barrier* function to indicate global synchronization of all active threads. It has the same functionality as pthread_barrier_wait, with an additional argument to

specify the synchronization barrier ID. Each unique global barrier calls sync_barrier with same ID. This step is performed by the programmer, to annotate the global synchronization points in the program. Note that the same barrier can appear at different locations in source code, as when different functions are executed by different threads. It could also be that the single barrier placed at a single location in source code is duplicated by compiler and appears at multiple instructions in the intermediate representation of the program. The barrier ID ensures that the compiler can still find the barrier points in all such scenarios. The ID allows the compiler to determine the barriers where all threads synchronize.

Next, we use algorithm 3, to incrementally identify these barriers starting from the beginning of a multithreaded section. It uses the RICFG (step 1) as created by algorithm 2. In the RICFG, a single, global sync node is created for each $sync\_barrier$ call with the same ID. The Disjoint-TS algorithm proceeds by iterating over each multithreaded thread section found by the Single-TS algorithm (step 2 and 2.1). It finds the reachable barrier nodes from start node through a depth first search (DFS) traversal of the RICFG, stopping at the sync nodes. If all the paths from the beginning of a thread section reach a single sync node (step 2.3.1 and 2.3.2), it implies that all possible paths that start from the beginning of the multithreaded code section have to go through the same barrier, and so this section can safely be demarcated as a new Disjoint-TS (2.3.2.1). The new common sync node is then used as a starting point for another search (2.3.2.2), to find the next Disjoint-TS. On the other hand, if the reachable set of sync nodes contains multiple elements, or is an empty set, then it conservatively abandons finding any new Disjoint-TS (2.1.3.3), and marks the end of the multithreaded section as end of current Disjoint-TS.

3.6 Overall Instrumentation Flow

The overall SBPA compilation flow proceeds as follows. We first perform an inter-procedural pointer analysis. Then, we build the RICFG. Then, Single-TS and Disjoint-TS algorithms are applied on the RICFG consecutively. Once we have detected the correct sets of memory locations accessed in different sections, we leverage per section modified/reference information to detect if the reads and writes in various sections can potentially share data with other threads. If a read/write is in single threaded code sections only, it is guaranteed to not have sharing issues. As the evaluation shows, we correctly detect nearly 80% of the single-threaded reads and writes.

For multithreaded code sections, we evaluate each read instruction as follows. If any write accesses the same points-to set as accessed in the read instruction in any of the thread sections where this read instruction can be executed, it implies that the memory accessed by the read could potentially be written in the same thread section. So the read must be instrumented. Otherwise, the memory accessed by this read is a read-only memory in the thread sections in which the read instruction is executed, and so the read is not instrumented. All thread sections in which the read instruction can appear are considered before the read can be determined as non-conflicting. Since single threaded sections do not overlap with multi-threaded sections, the writes in single threaded mode are automatically excluded from consideration.

Chapter 4

Programmer Annotations and MTROM

In this work, we focus on automatically reducing instrumentation without any program modification. However, in practice, due to various other overlapping compiler optimizations that can make the CFG more complex to analyze, and the pessimism of points-to analysis, sometimes static analysis fails to derive the most precise sets of conflicting loads and stores. In such cases, it is useful for the programmer to insert directives that help compilers understand the program flow more easily, The directives act as hints; inexpensive runtime checks are performed to verify that they hold true at runtime. The runtime checks are executed very infrequently compared to the frequency of instrumented memory accesses. We propose two types of directives: A directive to indicate multi-threaded read-only memory, and a pair of directives to mark the multi-threaded sections.

4.1 Marking Parallel Code Sections

Sometimes, thread spawn and join calls have conditional calls that make it difficult to automatically detect the fork/join pattern. For these cases, we propose a pair of directives `single_mode_begin` and `single_mode_end`. These directives do not change the behavior of the program, they just notify the compiler that the code section between the `single_mode_begin` and the first immediate post-dominator `single_mode_end` is executed by a single thread. To verify that the programmer does not introduce bugs, the runtime system verifies that when these are invoked, only a single thread is active. Otherwise, a runtime error is generated. These same directives can be used when the programmer knows that a single thread is performing work while the others are waiting. These were used only in 2 benchmarks in our evaluations.

4.2 Multi Thread Read Only Memory

In this section, we describe a memory model of Multithreaded Read-Only Memory (MTROM), that we define as a region of memory that is read-only, *only when* an application has multiple active threads. For data races to happen across multiple threads, at least one thread accessing a shared memory location must perform a write to that location while some other thread reads or writes to the same location in the same thread section.

Programming Patterns: It is quite common for parallel sections to just read a vast amount of data that is created in single threaded sections. A very common example of this is the parallel dense matrix multiplication. During the parallel phase of the program, the matrices being mul-

tiplied are not written to; only the product matrix is modified. Hence, the input matrices can be allocated from MTROM memory space. Another example of the usefulness of this technique is in the case of large graphs where additions and deletions only happen in single-threaded sections, but the query operations happen in multi-threaded sections. In the benchmarks we tried, many exhibited this nature of parallel execution. Thus, the MTROM model also aids the programmer understand the communication patterns easily in a multithreaded application.

All MTROM memory are allocated from a special heap, called *MTROM-heap*. This heap can be viewed as a segment of program heap separated by virtual address. At the beginning of multi-threaded phases, these heaps are marked as read-only through page fault protection available on most systems.

Theorem 1.: If all memory regions that a pointer variable can point to at a read instruction are MTROM memory locations, then that read instruction does not need to be instrumented.

Proof: Since it is guaranteed by the compiler, and additionally checked by the runtime system that MTROM memory is never written in multi-threaded sections, all the possible memory regions pointed to by this pointer are invariant in all disjoint-TS sections. Hence, there is no race or non-determinism through this access. So, excluding this read access from instrumentation is conservatively correct. Note that this access can only be a read, as a write will be an immediate violation.

Ideally, static analysis as described in chapter 3 would identify all such accesses. But due to limitations in pointer analysis and other static analysis methods in tracking large number of objects allocated at different places, it is not always possible to do so. In the MTROM

approach, the programmer allocates such memory from MTROM heap by replacing allocation routines with corresponding MTROM routines. The programmer can also mark some globals as MTROM, if desired. In practice, we observed that most large chunks of such memory are heap allocated or `mmaped`, as their sizes are not known upfront. We provide MTROM versions of functions (such as `malloc` and `mmap`), and require the programmer to use them when allocating MTROM memory. The pointer analysis then marks the memory objects created by these methods as 'MTROM heap', and similarly for global memories, it marks them as 'MTROM global'. During access instrumentation in thread sections, if a read memory access is traced to a points-to set that only contains MTROM-heap or MTROM-global regions, then that memory read access is not instrumented. On the other hand, if there is a write access to a points-to set containing only MTROM objects in some parallel section of the program, then the compiler pass can flag that as an error or warning. If the points-to set is a mixture of both MTROM and non-MTROM memory regions, then the read and write operations are instrumented as usual.

To detect incorrectly marked MTROM memory regions, a test can be be implemented either in all the executions or during debug mode. To avoid adding overhead with additional checks in the write logging functions, we protect the pages of MTROM heap by marking them as read-only. The read-only protection happens during multithreaded execution only, and are marked as read-write whenever the program goes to a single thread mode. These checks are to detect incorrect MTROM annotation by the programmer, and also to alert inadvertent writes to such memory locations in parallel mode.

In the benchmarks we analyzed, using MTROM required minimal changes (a few lines) to the source code. The only change was to replace few `malloc`, `mmap`, `free` and

`munmap` with their MTROM equivalents, and annotating at one place to indicate that the memory being accessed is MTROM. This method is not automatic, but it is robust and general, since the writes to such memory are protected in parallel phases. It provides a strict guarantee to the programmer that there is no communication (or races) through the MTROM memory regions.

In the past, various authors have proposed different memory modeling and allocation for multi threaded programming also. Berger et. al. [12] proposed Hoard, that maintains per thread heaps so that parallel, scalable memory allocation is possible, and false sharing is also easily eliminated for thread local memory. Our approach of MTROM is for memory shared among threads that cannot be made thread local. Adve et. al. [2] provide a perspective on various memory modeling for multi-threaded programming, that includes sequential consistency, Java and C++ memory modeling. The MTROM modeling proposed here is much simpler, and it provides a completely conflict memory, avoiding any of the issues associated with the models described in [2].

Chapter 5

Loop Invariant Log Motion

In this chapter, we describe few techniques to reduce instrumenting of memory accesses in loops. Loops in programs consume significant portion of the runtime, and hence any optimization in the loops is of great significance. This work and its results are also described in brief in our workshop paper [31]. We note that our optimizations described here has some similarity with the work described in REDCard [42].

Loop Invariant Code Motion (LICM) is a well known compiler optimization technique, that hoists code independent of the loop outside the body of the loop, either before or after the loop. A reaching definition analysis is usually performed to identify such code. Example 6 shows a small code excerpt illustrating such a case. The computation of $exp(x)$ can be moved to the loop pre-header, and the computed value can be used within the loop body.

Note that if the loop invariant code is as shown in example 6, then SBPA instrumentation pass automatically benefits from reduced instrumentation, as the loop pre-header is only

Example 6 A simple loop with loop invariant computation of exp(x). After LICM, the computation is moved into the loop preheader and computed only once, thereby reducing execution time

```
Original loop:

for(int i = start; i < end; i++) {
   C[i] = A[i] * exp(x);
}

After LICM:

v = exp(x);
for(int i = start; i < end; i++) {
   C[i] = A[i] * v;
}
```

executed once for multiple iterations of the loop body. However, for our purposes, we are only interested in set of accessed memory locations and not the actual values of those memory locations. So, to reduce instrumentation overhead, we extend the notion further. to only look at the pointers. So, there is potential for further optimization regarding this. Similar to LICM, we call it *Loop Invariant Log Motion (LILM)*.

We describe two types of Loop Invariant Log Motion in the next sections.

5.1 Scalar Loop Invariant Log Motion (SLILM)

Example 7 An example of repeated accesses in the loop through the pointer p. The value of $*p$ may change in each iteration.

```
for(int i = start; i < end; i++) {
   *p += C[i] ;
}
```

Consider the code in example 7. In this example, there is a store to the memory location pointed to by p in each iteration of the loop. So, the value of *p is different each time, and LICM may not move the operation outside the loop body. However, for the purposes of instrumentation, the pointer location p does not change from iteration to iteration. Hence, we only need to instrument it once for the entire loop. If p itself changes during loop execution, this optimization cannot be applied. Algorithm 4 shows the steps of applying SLILM to loops in a program.

Algorithm 4 Detecting Scalar Loop Invariant Log Motion.

```
1 foreach loop L in program P {
  1.1 If L has thread directive, skip L
  1.2 for each memory access a in L without an offset {
    1.2.1 Let p be the pointer value.
    1.2.1 If p does not need instrumentation, skip a.
    1.2.2 If there is any assignment to p in L, skip a.
    1.2.2 Remove any instrumentation for a if exists
    1.2.3 if a is store {
      Add logStore(p) in pre-header of L
    }
    1.2.4 if a is load {
      Add logLoad(p) in pre-header of L
    }
  }
}
```

5.2 Vector Loop Invariant Log Motion (VLILM)

SLILM as described above only works for scalar pointer accesses. It is more common to use the loop counter in array offsets to access memory locations from a loop invariant base pointer. In example 8, the read access of A[i] and B[i], and write access of C[i] all fall in

Example 8 A simple loop with reads and writes to illustrate VLILM

```
for(int i = start; i < end; i++) {
   C[i] = A[i] * B[i];
}
```

this category. These accesses cannot be hoisted outside the loop by LICM, as the values are different in each iteration. Since the pointer values (A+i), (B+i) and (C+i) are also different in each iteration, all these accesses must also be instrumented within the loop body.

However, our basic insight is that there is significant locality in the memory locations in such accesses and efficiency improvements can be achieved if we can consolidate multiple access instrumentations into a single instrumentation. The reason is that the instrumentation function execution is runtime intensive when it is invoked billions of times in critical loops. For example 8, we can naively instrument each of the accesses $C[i]$ for i ranging from $start$ to end. Or, we can call one instrumentation function with a range of $start$ to end and place it in the loop pre-header. Depending on whether the access is load or store, the instrumentation call will be one of these two listed below.

- $logStore(Type * address, size_t start, size_t end)$
- $logLoad(Type * address, size_t start, size_t end)$

Note that these vector versions of instrumentation functions are dependent on the element type, as the size of the element determines the step sizes of the offset. The logging function is optimized to stride through the array, since element size is known. Assuming that the array element size is S, the cache line size used by the runtime model is C, and the number of loop iterations is N, this will call the logging function $\lceil N \times C/S \rceil$ times. Without this

optimization, the logging function is called N times. If S is a 32-bit integer or float, and C is 32 bytes, this reduces the number of calls eight times.

Such optimizations can only be performed if the entire loop is within a single Disjoint-TS; so the loop cannot contain any thread related directives, such as a thread barrier, or task begin or end for STMs. Also, the base address of array must be invariant in the loop. This ensures that the single consolidated call is equivalent to the multiple original calls. Depending on the size of the array element, this can significantly reduce redundant calls. This is applicable to both read and write accesses. This optimization is most useful when we have unit step functions for the loop induction variable. With a small stride step function, this may still provide some benefit. A unit step appears to be the most common case in practice.

Note that there is a small difference in how these methods work for race detection and STMs, and how they are applied for deterministic runtime systems. For race detection, only the address is required. For STMs, only the original value of the memory location is required for a restart on conflict. But for deterministic execution, the final values of the memory locations are required. Therefore, for deterministic systems, the single SLILM and VLILM instrumentation is moved to the end of the loop instead of the preheader.

For nested loops, we only applied this optimization to the innermost loop. For example, in nested loops for matrix multiplication, the pass is only able to optimize the accesses to the elements when consecutive elements are accessed in successive loop iterations, which happen more frequently in the innermost loops. Although one could hoist an optimal number of logging operations outside of nested loops, we haven't explored that. Also, we only apply this optimization when the loop has no conditional jump that could exit the loop early. This is

required, so that the logging operation does not log accesses that may not actually be executed in the loop body.

5.3 Result of Applying LILM

We present the results of applying LILM here, as LILM is excluded in the main results presented in the next chapter. The main reason is that when we integrated our pass with ThreadSanitizer pass, we did not want to modify the ThreadSanitizer library. And so, we had no way to measure improvement with LILM and ThreadSanitizer. Therefore, we present the LILM results separately here.

We evaluate the reduction possible with SLILM and VLILM when combined with other SBPA optimizations. Hence, our base here includes the optimizations described in prior chapters. The column headings of the tables are explained below.

- No-LILM: This includes optimizations that are part of Coredet; namely successive access optimization (SAO) and Thread escape analysis. It also includes SBPA thread section analysis combined with modref analysis.

- With-LILM: Optimizations in No-LILM combined with SLILM and VLILM

Table 5.1 shows the reduction in the number of instrumentation calls to the runtime library with LILM and without LILM. Similarly, Table 5.2 shows the reduction in dynamic load instrumentation counts when using LILM with other optimizations. As seen in the tables, in many cases LILM failed to show any benefit. Also of interest is that LILM can produce significant benefit even in store instrumentation counts. Most of the other optimizations, as shown in

Benchmark	No LILM	With LILM
blackscholes	92.5	92.5
fft	52.4	52.4
lu	33.3	33.3
radix	44.9	63.5
histogram	66.8	66.8
kmeans	50.6	75.2
matrix_multiply	1	50
pca	99.8	99.8
reverse_index	0.8	0.8
string_match	58.6	58.6
linear_regression	62.0	62.0
Average	**51.17**	**59.55**

Table 5.1: Total access logging reduction improvement with LILM

chapter 6 mainly reduce read instrumentations. In many cases, the fact that the optimizations without LILM already reduced significant instrumentation (e.g. blakscholes) means there was not much scope for LILM to improve it further.

Benchmark	No LILM	With LILM
blackscholes	100.0	100.0
fft	30.9	30.9
lu	1	1
radix	38.5	61.4
histogram	50.3	50.3
kmeans	50.4	75.1
matrix_multiply	1	50
pca	99.9	99.9
reverse_index	1	1
string_match	100.0	100.0
linear_regression	100.0	100.0
Average	**52.0**	**60.8**

Table 5.2: Load logging reduction with LILM.

Chapter 6

Experimental Results

6.1 Experimental Setup

We implemented SBPA as an LLVM compiler pass for a software runtime system which enforces deterministic execution of multithreaded applications. The software runtime system is similar to CoreDet [11] and has to instrument loads and stores to detect conflicts, unless proven to be non-conflicting. We used CoreDet's optimizations as our baseline for a system that does not perform task logging. These optimizations include escape analysis to avoid instrumenting thread-local data, and elimination of redundant instrumentations for same memory address on successive accesses.

Our compiler pass is implemented for LLVM 3.3. We included the poolalloc [65] module with Data Structure Analysis (DSA) that uses Steensgaard alias analysis [92]. We enhanced it as described in section 3.2 to work with SBPA. We compiled each benchmark with Clang's -O4 optimization level to include link time optimization (LTO). Thereafter, we applied

SBPA analysis and instrumentation pass on the intermediate representation (IR) of the whole program.

To evaluate SBPA we used benchmarks from the PARSEC [14] [15], SPLASH [99], and Phoenix [86] benchmark suites which are shown in Table 6.1. We included results from all of the benchmarks that we are currently able to compile and run with our deterministic execution runtime system which is under development. At runtime, we recorded the dynamic counts of the number of loads and stores which were instrumented.

We modified the Phoenix benchmarks to allow us to specify the number of threads to spawn. For measuring dynamic instrumentation counts, all benchmarks were run with 2 threads, as the number of threads does not influence these counts. For comparing runtime improvement with ThreadSanitizer, as explained in chapter 7, we used 2, 4 and 8 threads. Unless otherwise stated, each benchmark is unmodified from its original state.

Suite	Benchmarks
PARSEC	blackscholes (bl), dedup (de), fluidanimate (fl), swaptions (sw), canneal (ca), streamcluster (st)
SPLASH	fft (ft), lu (lu), ocean_cp (oc), ocean_ncp (on), raytrace (rt), radix (rd)
Phoenix	histogram (hi), kmeans (km), linear regression (lr), matrix multiply (mm), pca (pc), reverse index (ri), string match (sm)

Table 6.1: Benchmarks used, abbreviation shown in parentheses.

Certain benchmarks in parsec were excluded due to inability to run these with multiple threads, even using parsecmgmt utility provided. The pthread configuration was not available

for freqmine and raytrace. However, a similar benchmark, raytrace, was included from splash suite. We were also not able to compile two other parsec benchmarks facesim and vips due to some complex manner in which these are compiled. However, we think the large set of benchmarks we included from 3 established suites represents the parallel benchmarks very well for our purposes.

6.2 Results

The goal of SBPA is to identify as many non-conflicting loads and stores as possible. A non-conflicting memory access is guaranteed to be data-race free, and it also does not require checking for conflicts in Software Transactional Memory (STM). Non-conflicting stores can be classified as log-only (LogOnly) for STMs to reduce overhead, or can be ignored for data-race detection. We compare SBPA results with the current state-of-the-art compiler technique used by CoreDet [11] that we call "Base" in this section. On top of Base, we add four improvements: Single-TS (Section 3.4), Disjoint-TS (Section 3.5), Improved Alias Analysis (Section 3.2) and Directives (Section 4).

The evaluation is divided into five sections: overall results, analysis of reduction in instrumentation, benchmark insights, compilation overheads and runtime with ThreadSanitizer 7. Overall results provides details about the effectiveness of SBPA. Benchmark insights explains the source of instrumentation and/or the reason SBPA failed to identify more memory accesses as non-conflicting. The evaluation finishes by showing that SBPA pass requires minimal compilation time, and that in fact, sometimes it makes the compilation process faster because it

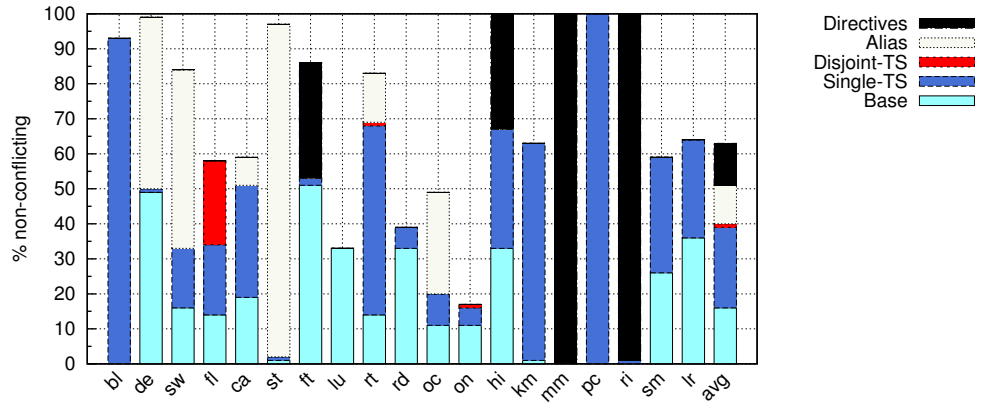

Figure 6.1: SBPA compiler pass detected 63% of all memory accesses at run-time as non-conflicting. Excluding the improvements from 'Directives' yields 51% accesses proven as non-conflicting.

reduces the amount of loads and stores that require instrumentation.

6.2.1 Overall Results

Figure 6.1 summarizes the results for SBPA. For each benchmark, it shows the percentage of non-conflicting memory operations that are identified as such at compile time. On average, 63% of the memory operations are proved non-conflicting. The current state-of-the-art (Base) only proved 16% of the memory operations as such. This is the dynamic count of loads and stores, and not the static counts. In all our plots, we show the dynamic counts of memory operations, not the number of loads or stores in the binary, as the latter does not represent runtime overheads.

The most effective technique is Single-TS. It is able to avoid checks for 22% of the memory operations. In some complex pointer cases, MTROM helped identify the truly invariant memory regions, and so some cases showed drastic reduction with that method. $Alias$ helped

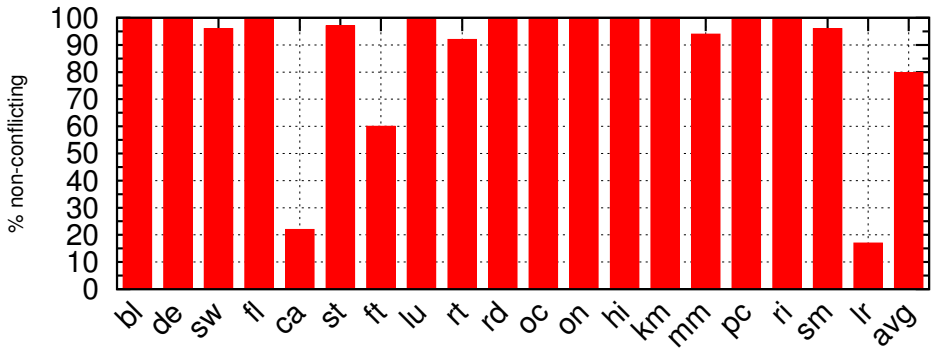

Figure 6.2: SBPA identified 80% of the dynamic memory accesses executed in single threaded mode.

disambiguate some pointer aliasing, thereby reducing 11% of the memory access instrumentations on average. The benefit is most apparent in 5 applications (de, sw, st, rt, oc) as the base alias analysis was creating alias between different thread sections. Since we can guarantee that some loads and stores cannot execute simultaneously, we were able to improve the precision of alias analysis. $Directives$ significantly affected 4 applications (ft, hi, mm, ri). Section 6.2.3 provides a detailed explanation per benchmark, but the main reason is the declaration of some variables as MTROM. On average, Directives removed 11% of the memory access checks.

From these benchmarks, only fluidanimate (fl) has a significant improvement due to Disjoint-TS. The reason is that very few benchmarks have different barriers during multithreaded code sections, or perform a significant percentage of work in each disjoint thread section. Instead, most applications use a spawn fork/join model where threads are spawned to perform a task, and then they are joined by the main thread. This kind of model is where Single-TS benefits the most.

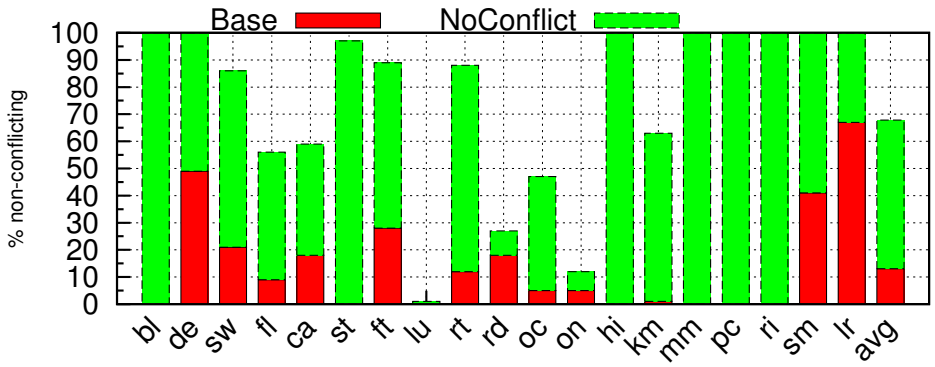

Figure 6.3: 68% of the loads are detected as non-conflicting, with a few applications reaching 100%.

6.2.2 Analysis of Reduction in Instrumentation

To measure the effectiveness of Single-TS pass, we ran the benchmarks with and without SBPA optimization, and measured the number of memory access instrumentations that happened when only one thread was active in the program. Figure 6.2 shows the percentage reduction of one over the other. A 0% reduction implies that SBPA couldn't prove any access in single threaded code sections as non-conflicting. A 100% implies that all such memory accesses were identified. On average, it identified 80% of these memory operations. Most of the benchmarks have over 90% reduction except ca, ft and lr. One of these (lr) actually has very few such memory accesses due to mmap operation for the input file, and all the data being read in multithreaded mode.

To understand SBPA behavior, we also show the effectiveness for loads and stores separately. Figure 6.3 shows the percentage reduction in instrumentation for loads when applying SBPA optimization pass. The state-of-the-art (Base) removes only 13% of loads on average. SBPA can remove an additional 55% of loads. This means that on average 68% of the load op-

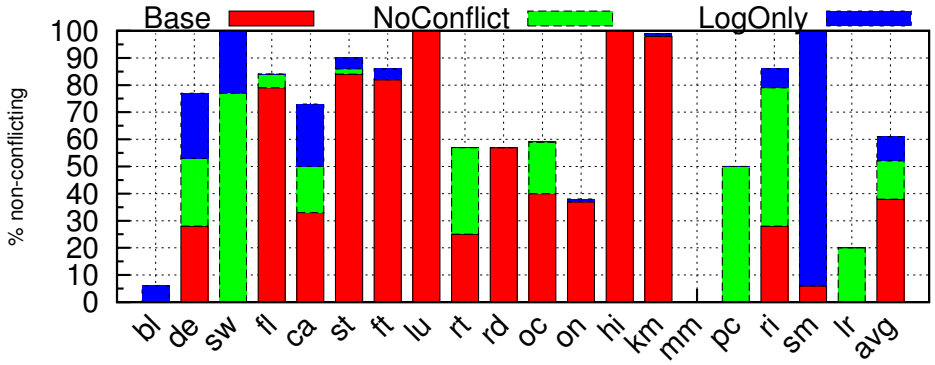

Figure 6.4: 61% of the stores do not need to be tracked by tools like data-race detectors which is much better than 38% with Base. For STMs that may have restarts, 52% of the stores can be proven as safe.

erations can be proven non-conflicting (NonComm). The results are better than those for the aggregate access counts because SBPA achieves a significant improvement for read-only accesses. The breakdown is also fairly disparate; for 8 benchmarks (bl, de, hi, pc, ri, mm, sm, lr) out of 19, it is able to remove over 99% of the loads that would need to be instrumented. This leaves little opportunity for other techniques to show any further improvement. The worst result is in lu, where nearly all the loads are marked as conflicting loads. The problem in lu is that the input matrix is modified in-place. The accesses are partitioned between threads in sub-blocks, but all access the same matrix. As a result, the alias analysis collapses all the memory accesses to a single points-to set. Even an ideal pointer analysis will have this problem, unless a range or partition analysis is performed.

Figure 6.4 shows the percentage reduction in instrumentation for all stores. In case of stores, we divided the reductions as Base, LogOnly, and NonComm. The state-of-the-art (Base) is able to mark 38% of the stores as non-conflicting. SBPA improves over that result, removing 61% of the stores for tools like data-race detectors. For STMs, SBPA proves over 52% of the

stores as non-conflicting.

$Alias$ also plays a part in these instrumentation reductions. For the benchmarks we evaluated, it helped to reduce 12% of load accesses for baseline, and 19% of loads for SBPA. Its effect on store reductions was minimal ($< 1\%$).

6.2.3 Benchmark Insights

To better understand these results, we proceed to provide a detailed per benchmark explanation of the SBPA compiler analysis.

blackscholes (bl): bl iterates over 4 global read vectors. SBPA is able to detect that most of the accesses are read-only, and reduces load access instrumentation by 100%. For store accesses, the same array is written by multiple threads. Without a sophisticated range analysis to partition the array statically using symbolic values, we cannot eliminate the store instrumentations. But overall, SBPA still determines that 93% of memory accesses are non-communicating.

dedup (de): This is a pipelined parallel processing benchmark. The core function with most of the read accesses is $FindAllAnchors$. With base optimization, the read accesses are already optimized by almost 50%, as there is significant local heap memory used within this function. When combined with Single-TS and our improved pointer analysis pass, it is reduced significantly more, to almost 98%.

swaptions (sw): In swaptions, a very high number of loads and stores in baseline version happen in the function

$HJM\_SimPath\_Forward\_Blocking$. Even though it allocates large portions of memory locally, due to aliasing with other global memory regions, instructions accessing the local mem-

ories are also treated as if they are accessing global memories, and hence get instrumented. Single-TS alone didn't help in this case. When combined with our modified pointer analysis, Single-TS provides the expected benefit in load and store tracking reduction in this critical function.

fluidanimate (fl): fl is a very good fork-join model parallelism, with many global barriers for the threads. It reads and modifies different fields of the structures in a large array of structures, and so we needed to also use an improved, field-sensitive mod-ref analysis in combination with Disjoint-TS to see significant improvement in this case. Our strided access recognition was specifically designed to handle such situations. In our own version of such simulation benchmarks, where the code was written more carefully to unalias non-overlapping reads and writes in same phase, we saw 66% reduction in load access counts instead of 56% shown here, for this benchmark.

canneal (ca): This is a simulated annealing benchmark used in chip design. A circuit graph is first created in single threaded mode, with different elements and their initial placements. In parallel mode, an annealing algorithm picks random pairs of elements to consider for swaps. At the beginning, it swaps elements even if it leads to an increase in the routing cost. but as time advances and the temperature cools, only those swaps that improve the cost are carried out. Single threaded mode shows a significant decrease, as the graph creation process didn't' require any instrumentation. Also, the field sensitive pointer analysis helped, since only the coordinates of each node in the graph are modified, and not their connectivity matrix, which is frozen once the graph is created.

streamcluster (st): Streamcluster is an RMS kernel, that clusters a large number of points in different clusters based on a pre-determined number of medians. In this benchmark, we observed several local memory allocations, and also noticed that the medians are computed in a separate array. This implies that most of the fields of the points are not modified. The field-sensitive modified-reference analysis significantly reduced the amount of instrumentation needed in this case, as only a particular field of the points is modified to assign it a new center, while the coordinates of the points are never modified but read a very high number of times.

FFT (ft): ft has a global array of input time domain points that is not modified. SBPA detects accesses to this as non-communicating, and reduces load counts by 30%. Our improved Steensgaard analysis reduces this further, by disambiguating memory regions pointed by various function arguments. FFT is one of the few benchmarks that benefit from Directives, in which we used a per thread array that effectively privatized the data before each FFT iteration.

LU (lu): In LU, threads modify the input matrix in-place with complex strides. Our methods do not show significant improvement, except for reducing some access tracking that happen in single threaded mode. The benefit comes from detecting all the memory initialization operations as non-communicating, that account for over 80% of the stores.

raytrace (rt): Our Single-TS reduced a significant 61% of total read and 49% of total write access instrumentation. When compared to baseline instrumentation, 56% read and 32% write instrumentation were eliminated by this alone. When we combined this with our more precise pointer analysis, the above savings were further improved to almost 98% reduction in read and 57% in store instrumentation. We observed a very distributed pattern of reads and writes across many functions, and many of those functions were called from multiple places with different

arguments. That is one reason why our modified pointer analysis had a good impact in this case. Disjoint-TS also identified 2 sections within the multithreaded program segment, but this only helped marginally.

radix (rd): rd performs in-place sorting of the keys. We see some benefit from avoiding instrumentation during initialization, but very little benefit from applying SBPA specific analysis on the multithreaded sections. Radix is one of the 3 worst benchmarks. Nevertheless SBPA still proved 40% of the memory accesses as non-communicating.

ocean_cp (oc): oc has a significant boost compared with the state of the art. The loads benefit mostly from the improved alias analysis; without it we see no benefit in loads. There are three main functions, jacobcalc, jacobcalc2 and laplacalc, which perform most of the memory accesses. SBPA pointer analysis unaliased some of the points-to sets due to calls to some major functions with different formal arguments. Some memory regions become read-only, thus increasing the number of non-communicating loads. Stores benefit from Single-TS analysis, independent of the improved alias analysis.

ocean_ncp (on): In this case, the threads access global memory in such a manner that our methods do not find significant optimization opportunities. Our methods still remove approximately 8% more read accesses than the baseline. The additional reduction in store instrumentation over baseline was negligible.

histogram (hi): The threads read values from global data, and increment counters each time they read a value. Histogram uses a large memory malloc for a single array and passes a section to each thread, which creates aliasing among the memory operations. We modified the memory allocation to use per-thread malloced memory. This improved the aliasing, resulting in nearly

100% non-communicating memory accesses.

kmeans (km): km has a very high load-to-store ratio, stores being only a small fraction of total accesses. It also uses a global array of points, so SBPA can determine several accesses as read-only. Due to other interfering optimizations, such as loop trip optimization that replicates loops with branches to handle trip counts of 0, SBPA algorithm is not able to identify the most precise points where the multi-threaded regions begin or end. Still, it is able to remove a significant portion of instrumentation calls.

matrix_multiply (mm): This is a classic case of multiple threads accessing large portions of read-only memory, where the input matrices are not modified at all during the multiplication process. The problem in this benchmark is that a single large array is created for all the threads. When the input matrices were marked read-only (MTROM), SBPA reduced 100% of the load instrumentations, but it still instruments all the multithreaded store operations.

pca (pc): This program has two separate threaded regions. In the mean computation region, it is reading matrix data, and writing only the means of each row. In the subsequent threaded region to compute covariance, mean is only read. Hence, in the second threaded section, both the matrix data and mean data are treated as read-only, and do not need instrumentation.

reverse_index (ri): In this case, large input data set is is modified in-place, to NULL-terminate the end of http links embedded in it. This rendered its memory read-write in multi-threaded mode, although in principle it is only used as input. When the link end point information was annotated separately using less than 10 lines of code change, this large input data became read-only in each Disjoint-TS. Then, SBPA was able to identify nearly all the memory accesses as non-communicating.

string match (sm): sm is an interesting benchmark that exhibits a property not seen in any other. Most of the stores can be proved as log-only. This means that for data-race detection, nearly 100% of the memory accesses do not need to be verified. For non-communicating memory accesses (Figure 6.1), 60% of the memory accesses were identified as non-communicating for STM-like systems, due to many store operations present in the identified Disjoint-TS.

linear regression (lr): In this case, there are successive loads of the fields of the same structure. As a result, the state-of-the-art (Base) optimizations could reduce load tracking significantly. SBPA is able to detect all the reads as read-only, marking 100% of the loads as non-communicating, with an overall 62% reduction in aggregate access instrumentation. There is a significant amount of stores in this case, which results in smaller reduction in total access tracking compared to load tracking reduction.

6.2.4 Compilation Overhead

In this section, we show that SBPA, incorporated as optimization passes in LLVM compiler, is not computationally expensive, which makes it quite practical. SBPA uses the Clang front end to create LLVM byte codes. These byte codes are optimized and linked to create one final byte code, which is converted to machine code executable. SBPA pass is run as part of *opt* after all the input files have been combined into a single byte-code file. The instrumentation functions are inlined for faster execution. These steps are very similar to those used by CoreDet [11].

Figure 6.5 shows the compile times of Base and SBPA normalized against compile times without SBPA or Base. SBPA for STM system is between 15% and 90% faster than

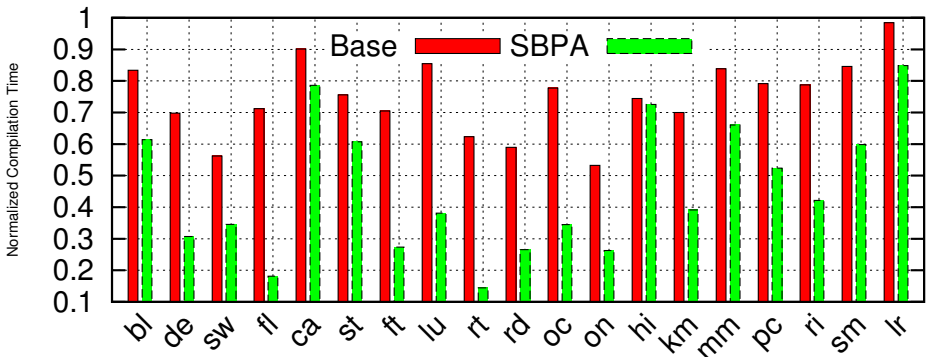

Figure 6.5: Compilation times of Base and SBPA normalized against compile time when no optimization is applied. On average, Base took 75% and SBPA took 46% of unoptimized compile time.

the same STM compiler pass without any optimization. In the benchmark with the highest overhead (on), the SBPA optimization pass accounts for less than 1% of the compilation time. For most of the benchmarks, it is less than 0.3% overhead. So, SBPA pass is runtime efficient, as even with small overhead of its own, it reduces overall compilation times for STMs and other deterministic engines, primarily because the instrumentation overhead for compiler is reduced significantly. Base optimization also reduces compilation time over unoptimized version. The extra computation time required for Base and SBPA is significantly less than the extra cost incurred in instrumenting the additional loads and stores. The second bar shows that SBPA reduces compile times significantly more than Base for every case.

For ThreadSanitizer experiments, we used a slightly different flow as described in Chapter 7. In this case, SBPA removes the unnecessary instrumentations already inserted by ThreadSanitizer. ThreadSanitizer instrumentation calls are external library functions and not inlined, so SBPA adds some compile time overhead in this flow. In our experiments, we saw a maximum slowdown of 42% in compile time compared to that of ThreadSanitizer pass. Still,

for many benchmarks we observed shorter compile times due to reduced linking times. The geometric mean slowdown was just 1%. We think, a small slowdown is acceptable, considering how much runtime can be saved, particularly when many tests are run with the same executable.

Chapter 7

A case study with ThreadSanitizer

To evaluate the effect of SBPA on runtime for dynamic race detection, we combined SBPA optimizations with ThreadSanitizer [89], a state of the art dynamic race detection tool. ThreadSanitizer works within LLVM framework, and uses Clang's front end to insert the instrumentation instructions. This is also one primary reason to choose ThreadSanitizer over other such race detectors. According to ThreadSanitizer documentation, it causes 5X to 15X slowdown, and anywhere from 3 times to 9 times more memory compared to executables without ThreadSanitizer. We let ThreadSanitizer instrument the binary as it deems suitable. Thereafter, SBPA is applied on the whole program model to remove the unnecessary instrumentations inserted by ThreadSanitizer. In this method, our pass has very low interactions with ThreadSanitizer instrumentation process. We evaluated each benchmarks with the following options.

- No-Tsan: ThreadSanitizer pass not applied. This is the native run, with no race detection.
- Tsan-Zero: ThreadSanitizer pass is applied, then all instrumentations are removed.

- Tsan: ThreadSanitizer instrumented; none of the Base or SBPA optimizations applied
- Tsan-Base: Apply Base optimization on Tsan
- Tsan-SBPA: Tsan with SBPA, that includes Base, Single-TS, Disjoint-TS and Alias
- Tsan-Directives: Tsan with SBPA, and annotations where applicable

Tsan-Zero is included to measure the overhead that remains even when all instrumentations are removed. This is not a correct optimization to do, but it provides a trivial lower bound of the runtime that can be achieved by removal of instrumentations. As we observe, some benchmarks have a significant overhead regardless of the number of instrumentations, while some others have low fixed overheads.

Tsan-Base includes the optimizations that we described in previous chapter, except certain optimizations that are not applicable to precise data race detection, and are dependent on the cache line size used in our previous chapter. Instead, we included an additional optimization, to exclude accesses that are enclosed by the locking and unlocking of the same mutex variable. This is somewhat similar to the *Unlocked Pairs Computation* technique in [74], except that we can only remove an instrumentation when it is guaranteed that such removal will not miss any race. So, instead of may aliasing, our checking is based on must aliasing of the lock variables. If a memory location is accessed only in lock protected regions that are all protected by the same lock, then we remove instrumentation of accesses to that memory location. Although this technique should theoretically yield improved results, in reality only a few accesses are made in such critical regions due to the overhead of locking and unlocking, and because it inherently causes serialization during parallel execution. Secondly, if the mutex variable is not global, then

it is difficult to prove statically that memory regions are indeed protected by the same mutex. In our benchmarks, we only observed minimal benefit by using this optimization.

Test	No-Tsan	Tsan-Zero	Tsan	Tsan-Base	Tsan-SBPA	Tsan-Directives
bl	90.3	107.4	174.1	173.1	112.9	113.1
de	26.4	66.2	100.6	100.7	69.9	67.5
sw	22.0	32.5	180.8	171.1	48.9	148.9
fl	6.5	12.4	105.4	97.2	69.0	69.3
ca	28.7	119.3	153.8	151.5	119.7	118.6
st	11.3	11.8	281.9	277.8	21.3	21.3
ft	7.6	7.6	57.0	40.7	32.7	32.2
lu	1.3	1.4	71.9	60.4	53.2	53.2
rt	10.2	14.9	142.3	136.1	17.2	17.2
rd	27.5	27.7	59.5	60.0	57.7	57.5
oc	20.4	23.6	271.1	257.6	228.0	228.0
on	45.5	51.9	578.6	561.2	546.9	547.6
hi	0.9	0.9	29.1	22.4	11.9	7.9
km	26.6	28.9	440.4	440.6	245.5	245.4
mm	3.5	3.5	16.5	16.4	16.5	3.5
pc	25.5	27.1	939.6	936.5	28.0	27.9
ri	1.8	7.6	10.4	10.5	10.4	7.0
sm	5.1	155.3	213.2	211.3	196.4	196.6
lr	1.1	1.1	15.4	15.4	1.1	1.1

Table 7.1: Runtimes (in seconds) of executables compiled for ThreadSanitizer dynamic race detection with 2 threads.

Table 7.1 shows the absolute runtimes of the benchmarks executed under controlled load condition on a machine with 3.5 GHz Intel Xeon processors. Wherever possible, we used a large input data set to have a long enough runtime for the comparisons to be meaningful. The

large data inputs were maintained locally on the machine to get accurate runtimes that do not depend on the network speed.

In Figure 7.1, we show the speed-up for each benchmark with $Tsan - Directives$, which includes all the optimizations. These are compared to the runtimes of $Tsan$, as measured with 2, 4 and 8 threads. Some benchmarks have higher speed-ups as number of threads are increased. We think, it is because the instrumentation operation is performed as a serial process in most systems, since it needs to update data structures common to all the threads. So, once we remove an instrumentation, it becomes faster if there are more threads running in parallel. However, in some cases with lot of races, the race detection and reporting dominate the total runtime. In those cases, even removing the instrumentation didn't help the runtime significantly, regardless of the number of threads. In several of these cases, ThreadSanitizer reports races. And in all of those cases, the races continued to be reported after our pass was applied. This shows that our pass is working as expected, and that it is not removing any instrumentation that is necessary to detect races.

We also show the overall speed-ups achieved with different optimizations in Figure 7.2. These are the geometric means of speed-ups of all the benchmarks for NoTsan, Tsan-Zero, Tsan-Base, Tsan-SBPA and Tsan-Directives compared to *Tsan*. (The prefix "Tsan-" is removed in the figure for brevity.) Overall, SBPA and Base achieved speed-ups of 2.425 and 1.116 respectively on geometric mean basis. When combined with directives, the speed-up increased to 2.736 times. The upper bound of speed-up is 6.5 as with *Tsan-Zero*. We note that *Tsan* is about 12 times slower than *No-Tsan* for these benchmarks.

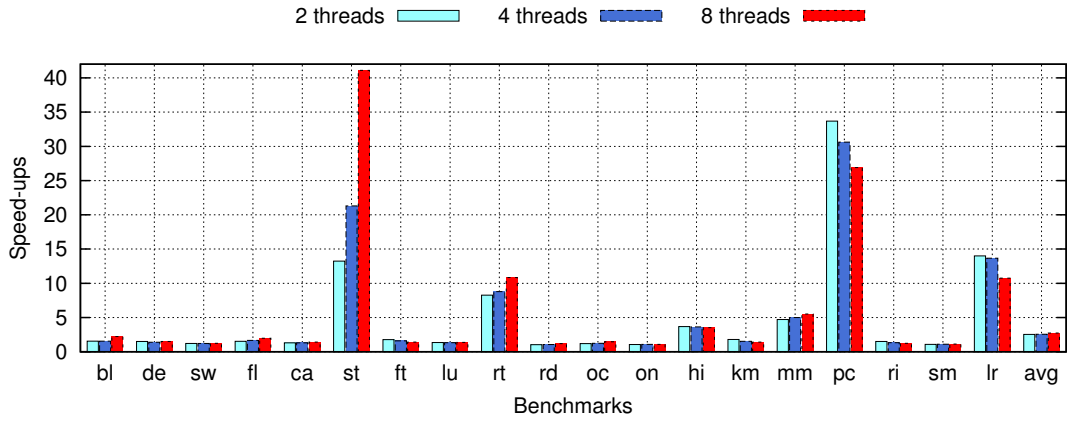

Figure 7.1: SBPA yields over 2 times speedup compared to Tsan, with some applications achieving over 30 times speedup.

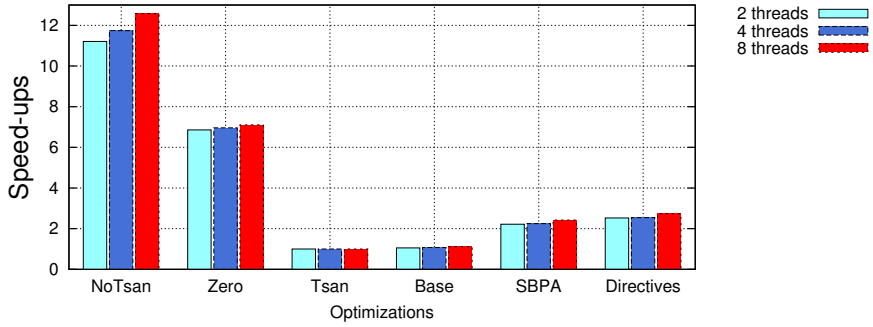

Figure 7.2: ThreadSanitizer speed-ups in the different modes described earlier. SBPA, combined with Directives, speeds up ThreadSanitizer execution by a factor of 2.74.

In general, the speed-ups correlate fairly well with the percentage reductions in instrumentation we showed in previous chapter. But we note that the improvement in runtime is not as high as the reduction in dynamic instrumentation count in some cases. For certain benchmarks, such as de, runtime overhead didn't decrease as the optimizations used for our STM as part of Base was specific to cache line size used for that system. On some other benchmarks, such as st, pc, rt, mm and lr, we see tremendous improvements. Yet, in some others, the

overhead even with zero instrumentation is quite high and dominates the runtime; so we do not observe as much speed-up as would be indicated by the reduction in instrumentation counts.

Chapter 8

Improving Static Race Precision with SBPA

A very important issue in writing multi-threaded programs is to identify potential trouble spots during developing the multithreaded software. This is the subject of *static race detection*, where the compiler or the static race detector flags potential racy accesses. Note the difference with dynamic race, which detects race conditions only at runtime and detects real races among threads at some memory location. In contrast to dynamic race detectors, static race detectors flag *potential* races, although in some rare cases, they can identify definite races. The reason is that it is very difficult to determine during program compilation, whether a race will actually occur, as it depends on inputs with which the program is executed. Due to the inherent imprecision of alias analysis and conservativeness of static race detection, static race detectors generate many false positives; i.e. they indicate potential races where there is no possibility

of actual race. This is a big difference with dynamic race detection. This is also precisely the reason that such tools remain impractical for use in real programs.

In this extension of our work, we apply SBPA to static race detection (vs. dynamic race detection as in prior chapters) and evaluate if SBPA can reduce the false positives, and thereby reduce programmer involvement to debug these unnecessary issues flagged by static race tools. We apply the same techniques and options as discussed in earlier chapters. However, instead of measuring dynamic counts of load and store instructions potentially involved in races, we measure the static counts. The objective of static race is to help the programmer fix the racy conditions. Note that SBPA operates on the Intermediate Representation (IR) of the LLVM compiler. So, we also map back the racy instructions in IR to the line in original source code, so that the programmer can relate to the issue much more easily. Therefore, for our evaluation, we measure the racy read and write instructions, as well as the racy read and write lines of program.

8.1 Methodology

For measuring static race, we only need to compile the benchmarks and not run it with any specific input. Similar to SBPA for dynamic instrumentation reduction, each source file is first compiled into individual byte codes. Then, these byte codes are combined and linked into a single byte code (IR) that represents the whole program. Then, LLVM *opt* is applied on this combined IR, SBPA is applied as part of opt. Various techniques are turned on with options specified to opt. For static race, we do not need to instrument the load and store instructions. Instead, an analysis is performed to detect the racy accesses used in reducing dynamic race.

And the various statistics are calculated. The racy lines of programs are printed out for the user to take further action.

To reduce the number of issues flagged, for each value accessed in source code, the issue is flagged only once. For example, if a global escaping memory location $V[i]$ is accessed few times in a function F, we only issue warning once for the value V in F. This reduces the number of issues the developer has to inspect.

We observe that some programs are intentionally racy for reasons of efficiency. In some cases, the race conditions never cause different results, as multiple threads write exactly the same value to a specific memory location and there are no inter-dependencies. In some other cases, the difference in values may be insignificant and immaterial. For example, there are well known implementations of graphic image compression utilities that are actually racy. So, there exists a class of race conditions that can be called *benign*.

So, the SBPA approach for static race adds flexibility for the user. SBPA compilation detects and issues warnings for the potentially racy accesses. The user can then verify the data access. If the programmer is confident that the access should never participate in a race, then he can annotate it in three different ways:

- **safe**: This access is safe to ignore. It could be a false race flagged, or could be a benign race. Although the user deems it safe, there is a possibility for race to occur, or the program output to differ due to race through such accesses. Dynamic race detection could still instrument these accesses in test mode, and capture differences in values over multiple runs of the same test.

- **critical**: This could be a critically important access, and should be verified further with dynamic race verification. Dynamic race detection then should ensure that the tests run with dynamic race detection has covered this access sufficiently.

- **check**: The programmer is simply unsure of the importance of this race, but he cannot rule it out either. So, dynamic race detection should insert instrumentation for this, and flag the race for this with normal severity. This is the default, and does not require annotation.

We believe that the reduced set of violations flagged by SBPA static race makes it feasible for the programmer to annotate the few racy accesses properly. And then, combining these annotations with a dynamic race detector that understands these annotations can provide very valuable insights into the race issues in the program.

8.2 An Example Case

In example 9, we first show the output from SBPA analysis. And then show the lines of code that it points to, along with the variable name (value). Since it operates on IR, the variable name is sometimes not exactly correct at this time. We plan to enhance this in future. The first location is a write to the thread ID array. Since the for loop encompassing this write also create threads, this line is assumed to be in a multi-threaded code section. However, we can see that this write could easily be moved into a separate loop executed before the loop that creates the threads. The programmer can fix this by rearranging the thread, or mark it as a *safe* race if he is certain. The other racy access pointed to here is the write to the global array prices at line 435. SBPA thinks that it is a potential race. However, the programmer might conclude

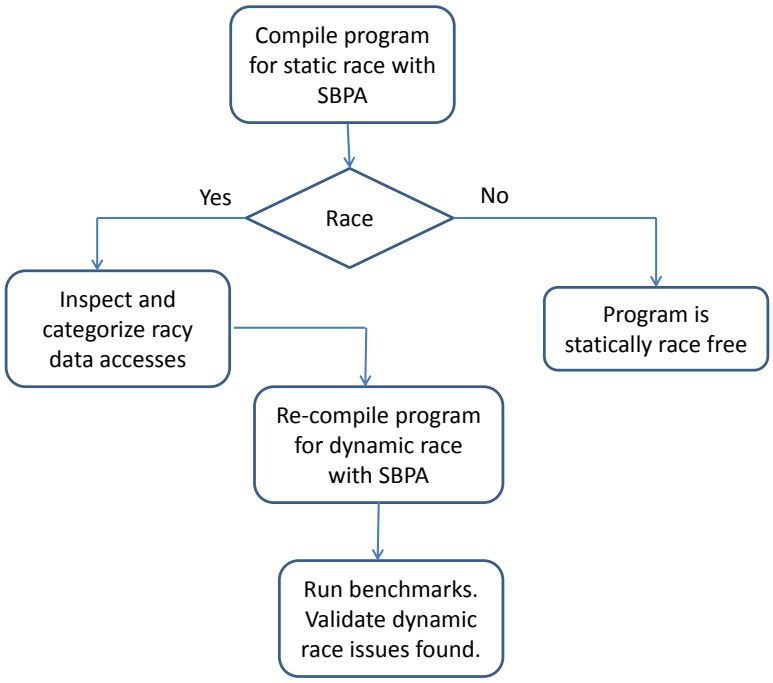

Figure 8.1: Static race detection flow with SBPA. Static races detected are classified and further validated by dynamic race detection.

that no two threads should write to the same location in the array, and can mark this as *critical*, which then gets verified by dynamic race detection accordingly.

By contrast, example 10 shows the many false positives when we do not use SBPA. Only two of the 14 potentially racy accesses were not proven by SBPA as non-racy; rest 12 were proven non-racy. Example 11 shows the code in program that cause most of these races when we do no use SBPA. In this particular case, the other global pointers are read only in the multi-threaded code sections, so these accesses are proven race free by SBPA. Note that the race detection is for each disjoint thread section. And the report also indicates how many writes and reads are involved in the race, and their locations.

Example 9 The potentially racy write access shown by SBPA race detector.

Output from SBPA race detection pass:

```
REPORTING RACES with 2 escaping write nodes out of total 42 (read nodes 13)
  Write Locations:
  File blackscholes.cpp:435
  Value is tids root value: tids
  Impacts 1 read locations in same DTS
==================================================
  Write Locations:
  File blackscholes.cpp:294
  Value is prices root value: prices
  Impacts 0 read locations in same DTS
==================================================
```

Snippet of source code

```
...
294 prices[i] = price;
...

433 int tids[nThreads];
434 for(i=0; i<nThreads; i++) {
435   tids[i]=i;
438   int _M4_i;
439   for ( _M4_i = 0; _M4_i < MAX_THREADS; _M4_i++) {
440     if ( _M4_threadsTable[_M4_i] == -1)    break;
441   }
442   pthread_create(&_M4_threadsTable[_M4_i],NULL,
          (void *(*)(void *))bs_thread,(void *)&tids[i]);
446 }
...
```

SBPA operates on LLVM IR, and therefore sometimes the variable names (known as *Value* in LLVM) don't always indicate the original variable name in source code. LLVM also sometimes creates additional variables as the IR is in SSA form. Besides, the value in a function could have been passed through the function argument or as a pointer in a structure passed to the function, and really points to some other location. That root location is indicated by the field *root value* in the output shown in examples 9 and 10.

Example 10 Output with base optimizations, without using SBPA

```
REPORTING RACES with 14 escaping write nodes out of total 42(rd nodes 16)
File blackscholes.cpp:412
Value is volatility root value: volatility
Impacts 2 read locations in same DTS

File blackscholes.cpp:416
Value is otype root value: otype
Impacts 2 read locations in same DTS

File blackscholes.cpp:420
Value is arrayidx98 root value:
Impacts 6 read locations in same DTS

File blackscholes.cpp:422
Value is arrayidx108 root value:
Impacts 7 read locations in same DTS

File blackscholes.cpp:423
Value is arrayidx113 root value:

File blackscholes.cpp:373
Value is prices root value: prices

File blackscholes.cpp:435
Value is tids root value: tids

File blackscholes.cpp:411
Value is rate root value: rate

File blackscholes.cpp:410
Value is strike root value: strike

File blackscholes.cpp:343
Value is nThreads root value: nThreads

File blackscholes.cpp:361
Value is nThreads root value: nThreads

File blackscholes.cpp:413
Value is otime root value: otime

File blackscholes.cpp:409
Value is sptprice root value: sptprice

File blackscholes.cpp:372
Value is data root value: data

File blackscholes.cpp:419
Value is arrayidx93 root value:

File blackscholes.cpp:294
Value is prices root value: prices
```

Example 11 Snippet of source code for the racy accesses.

```
393     _M4_numThreads = nThreads;
394
395         int _M4_i;
396         for ( _M4_i = 0; _M4_i < MAX_THREADS; _M4_i++)
397             _M4_threadsTable[_M4_i] = -1;
398
399
400 ;
401 #endif
402     printf("Num of Options: %d", numOptions);
403     printf("Num of Runs: %d", NUM_RUNS);
404
405 #define PAD 256
406 #define LINESIZE 64
407
408 buffer = (fptype *) malloc(5 * numOptions * sizeof(fptype) + PAD);
409 sptprice = (fptype *)(((unsigned long long)buffer+PAD)&~(LINESIZE-1));
410 strike = sptprice + numOptions;
411 rate = strike + numOptions;
412 volatility = rate + numOptions;
413 otime = volatility + numOptions;
414
415 buffer2= (int *)malloc(numOptions * sizeof(fptype) + PAD);
416 otype= (int *)(((unsigned long long)buffer2+PAD)&~(LINESIZE-1));
417
418 for (i=0; i<numOptions; i++)
419    otype[i]      = (data[i].OptionType == 'P') ? 1 : 0;
420    sptprice[i]   = data[i].s;
421    strike[i]     = data[i].strike;
422    rate[i]       = data[i].r;
423    volatility[i] = data[i].v;
424    otime[i]      = data[i].t;
425
```

8.3 Results

To measure effectiveness of SBPA in static race detection, we used the same set of benchmarks that we used to detect the impact of SBPA in reducing dynamic load-store instrumentation. But the counts here are all static counts; i.e. number of lines in source code or number of instructions in the executable. The various plots in this section show the benefit of using SBPA to reduce potentially racy accesses.

We notice in the results that the effect of some SBPA options is more subdued in many tests compared to their effect on dynamic instrumentation counts. It is because a single memory access in code can be executed many times at runtime. For example, the memory accesses within the loop of *matrix_multiply* are only counted as single access in these plots, though there are many dynamic accesses from that single access. When SBPA eliminates all these dynamic accesses, the reduction is only seen as one less access in these plots showing static access counts.

In Figure 8.2, we show the reduction in potentially racy read and write lines in source code achieved by `Base` and `SBPA` techniques. The higher the percentage, the more is the reduction and fewer remaining potentially racy lines to be investigated by the programmer. Similarly, Figure 8.3 shows the reduction in percentages of potentially racy read lines in code, and Figure 8.4 shows such reduction percentages for write lines in source code. Since SBPA actually operates on IR at the instruction level, we also show similar percentage reductions for read-write instructions, read instructions and write instructions in Figure 8.5, Figure 8.6 and Figure 8.7 respectively.

In Table 8.1 we show the absolute number of lines of code that SBPA shows as potentially racy. While it does very well in benchmarks like bl, lr, sm, mm and sw, it is less effective in some other benchmarks like oc and rt. The main reason is the imprecision of alias analysis, which is a common problem we observed in our dynamic race analysis.

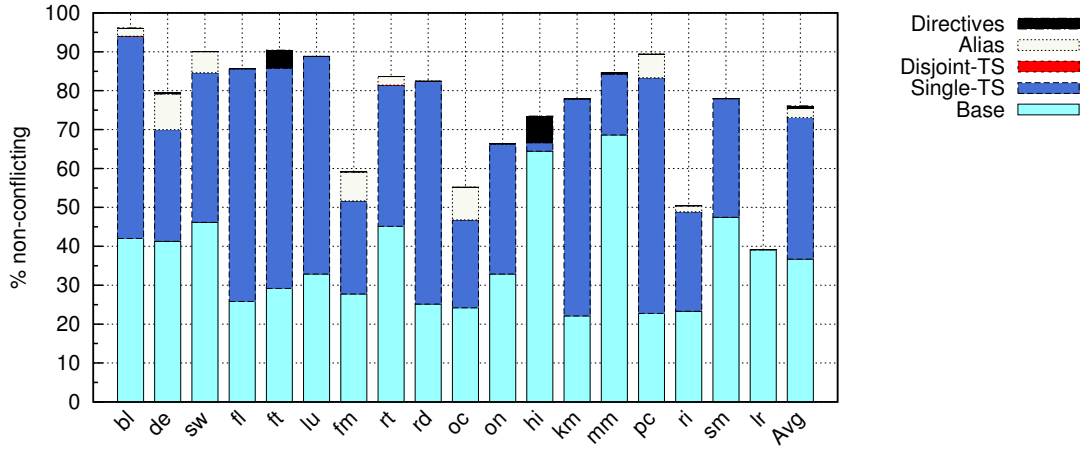

Figure 8.2: Percentages of total read and write lines identified as non-racy by Base and SBPA techniques.

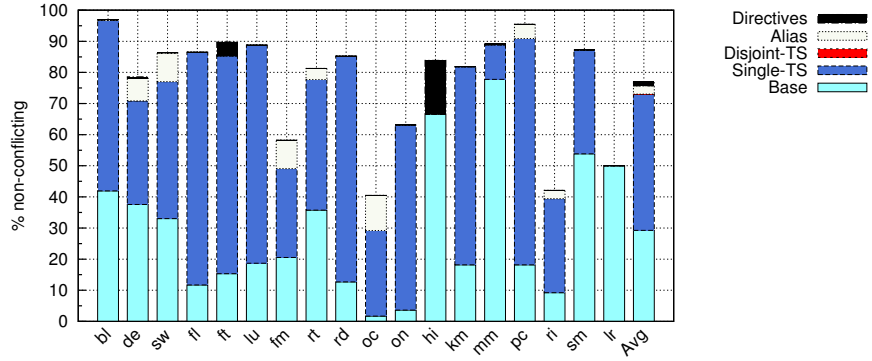

Figure 8.3: Percentages of read lines identified as non-racy by Base and SBPA techniques.

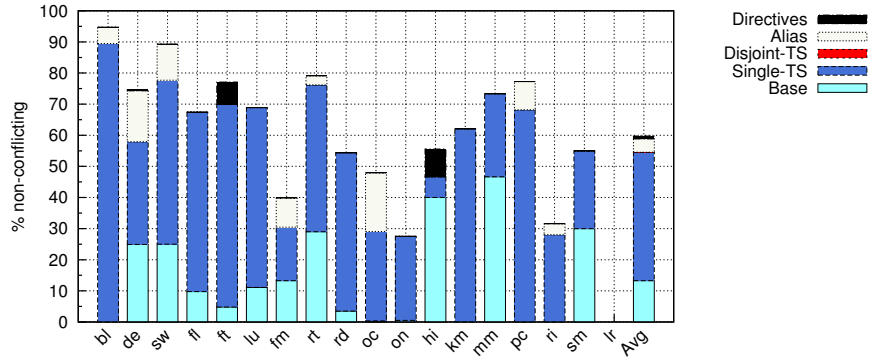

Figure 8.4: Percentages of write lines identified as non-racy by Base and SBPA techniques.

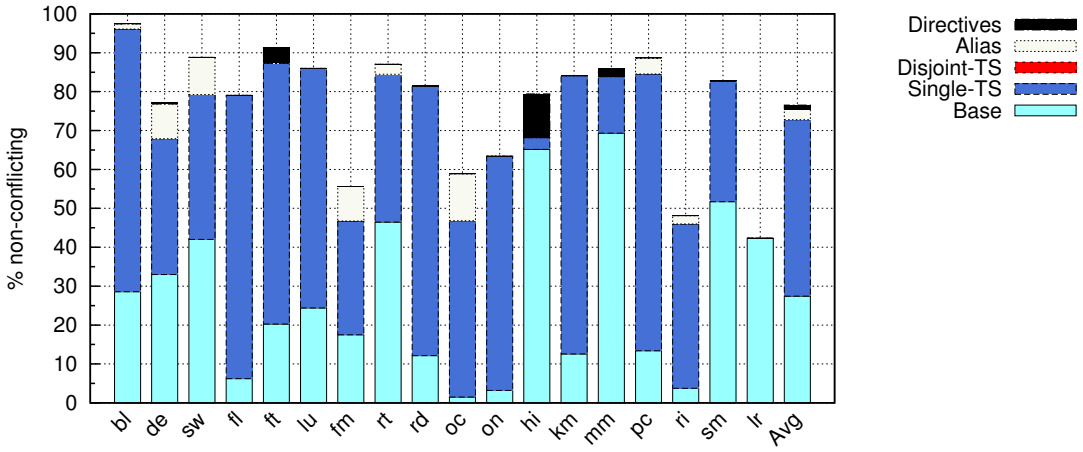

Figure 8.5: Percentages of read and write instructions identified as non-racy by Base and SBPA techniques.

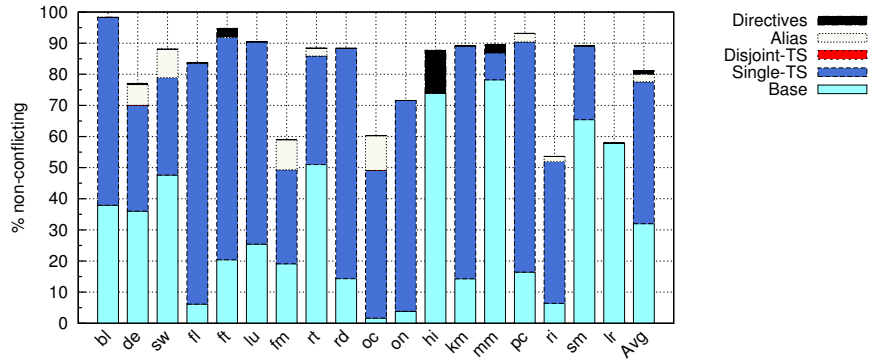

Figure 8.6: Percentages of read instructions identified as non-racy by Base and SBPA techniques.

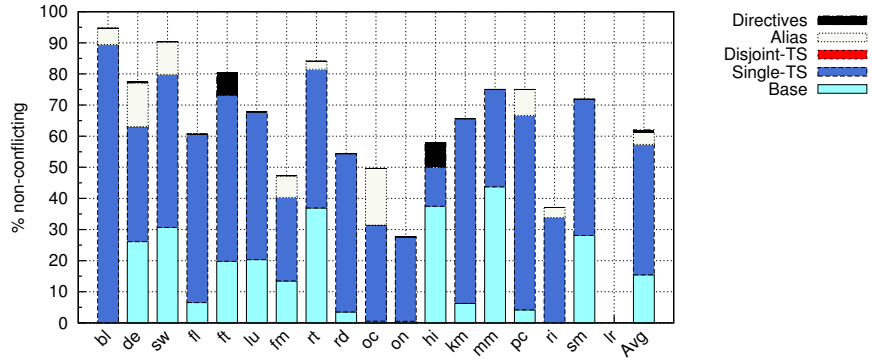

Figure 8.7: Percentages of write instructions identified as non-racy by Base and SBPA techniques.

Test	Number of racy read lines	Number of racy write lines	Total lines
bl	1	2	526
de	85	62	3084
sw	15	12	1554
fl	30	30	2815
ft	16	19	1051
lu	12	14	925
fm	289	154	4193
rt	199	152	9898
rd	21	26	1141
oc	887	279	5169
on	253	274	3265
hi	5	8	320
km	12	11	386
mm	4	4	448
pc	2	5	430
ri	44	39	639
sm	5	9	459
lr	8	7	324

Table 8.1: Total racy read and write lines, and total program lines for the benchmarks studied.

Chapter 9

Conclusion

In this work, we investigated several optimizations and techniques to reduce instrumentation overhead for shared memory multi-threaded programming. We conclude that a section-based static analysis of programs combined with suitable user directives, can significantly reduce the instrumentation overhead for race detection, STMs and other systems that detect unsynchronized memory accesses in multithreaded programs.

The most effective technique was Single-TS, which identified single-threaded sections of code in a multithreaded application. One reason for this is that the benchmarks were not written to harness the global synchronization barriers. As we mentioned in Section 6.2.3 for $fluidanimate$, when we write programs consciously to harness the global barriers, the effect from Disjoint Thread Sections within the multi-threaded sections is more significant. The programs written for GPGPUs are extreme examples of such globally synchronized programs, but have the additional limitations that all threads must execute the same code. Our proposed

barrier identification method does not require the code executed by each thread within such Disjoint Thread Sections to be identical, and paves the path for extending the global barrier notion to general shared memory multi-threaded programming.

Overall, SBPA eliminated a much higher percentage of load and store instrumentations compared to the baseline. Our ThreadSanitizer results show that SBPA can be used to improve performance of tools such as data-race detectors, STM and deterministic execution systems. We also applied SBPA to static race detection, and showed that it can be greatly beneficial in reducing false positives.

In some cases the fact that Single-TS removed nearly all of the instrumentation meant that there was no opportunity to demonstrate the potential of other techniques. Additionally, limitations in pointer analysis prevented the compiler from proving some instrumentation as redundant, In this work, we also made improvements to alias analysis that increased the effectiveness of the SBPA pass. that improved the effectiveness of SBPA. However, there is more which could be done, such as applying range analysis to partition pointers that appear to access the same memory regions, when in fact the accesses are disjoint. This is a topic of future work for us. Ultimately, we think the problem of detecting unsynchronized accesses requires a multifaceted approach of combining programming model changes, compiler support and runtime support.

Chapter 10

Future Work

In this chapter, we briefly discuss some possible future extensions of the work presented in this thesis. These are exploratory ideas, and may not be feasible to achieve. We think these have a good potential of reducing the difficulty of detecting and correcting unsynchronized thread communication further.

10.1 Symbolic Array Partitioning

One major issue of imprecise aliasing in array accesses is that at compile time, SBPA cannot determine the regions of the array each thread accesses. For example, in the case of $matrix\_multiply$, different rows of the output array are written by different threads. Presumably the rows are divided among multiple threads without any overlaps. However, lacking a symbolic analysis of this partitioning, it is impossible to determine this during program compilation. Note that such analysis necessarily needs to be symbolic, and abstract syntax tree level

information may be needed to understand the symbolic ranges assigned to each thread. If such an analysis can be added to SBPA, it can greatly enhance its precision for programs using large shared arrays. Most of the benchmarks partitioned the arrays in a simple manner, like assigning a range of contiguous rows to a thread. But in some cases, as in lu (LU decomposition), such partitions were more complicated, as sub-blocks of a matrix are assigned to each thread. A symbolic range analysis may be able to prove that the accesses to these partitions are race free.

10.2 Generalized SBPA and MTROM

In our earlier discussions, we have performed a top down analysis of the whole program to identify top level thread sections. Also, for MTROM, we have assumed that the program has only a single thread executing when such memory is written. In this section, we show that these ideas are extensible to more general cases, where spawned threads themselves spawn additional threads. Such situations are common in large applications, where a task executing as a thread itself spawns more threads to run faster. Furthermore, we show that it is feasible to implement the check for MTROM access in hardware, to speed up such checks to run at almost same speed as uninstrumented accesses.

10.2.1 Extension of SBPA to Tree of Threads

To extend our ideas of SBPA and MTROM to such scenarios, we consider a *tree of threads* model of threaded programming. Figure 10.1 shows an example tree of threads for a large-scale program. We denoted the root thread of the main user program as T. T in turn

creates two threads, *T1* and *T2*. So, *T1* and *T2* are children of T and T is their parent. *T1* spawns 3 threads *T11*, *T12* and *T13*. Each level of thread creations can be considered as different thread sections, which might in turn be composed of lower level thread sections. SBPA analysis can be extended to such hierarchies of threads, with each such analysis running in a sub-tree of the hierarchy. A bottom up analysis should be possible.

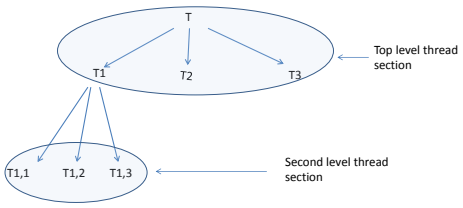

Figure 10.1: A hierarchy of threads in a program, represented as a tree

10.2.2 MTROM for Tree of Threads and Dynamic MTROM

Generalizing the idea described in 4.2, if a memory that is local to a thread section is not written by any thread other than the root thread of that thread section, then accesses to such memory do not need to be instrumented. Our earlier treatment assumed that an MTROM memory remains MTROM in the whole program. We think that it is also feasible to dynamically change an MTROM into multi-thread read write (MTRW) and vice versa with very low runtime overheads. The changeover can be executed between different disjoint thread sections, protected by global lock, to ensure correct operation. These dynamically changed MTROM can still benefit from static instrumentation removal, since the static removal can still assume that MTROM property is verified for the pointers via infrequent runtime checks.

In addition to dynamic transition between MTROM and MTRW, we could think of extending MTROM concept to a *tree of threads* system as well, by defining an MTROM ownership table. This table indicates which thread owns the write permission of the MTROM memory. It must be the root thread in a sub-tree of the thread tree where the MTROM memory is allocated. So, if *T1* allocates an MTROM heap, only *T1* can write to it and free it, and all its children threads *T11*, *T12* and *T13* can read from it. The MTROM ownership table can be implemented as a map of page number to thread ID.

10.3 Hardware Implementation of MTROM

A hardware page ownership table for threads could speed up the checks performed for MTROM detection. Three tables can be used to implement such a scheme: $PageOwner$, $ParentThread$ and $ActiveThreads$.

The table $PageOwner$ has as indices the page number of a memory page, and maps the page number to the thread id of the page owner. So, *PageOwner[page number]* indicates the thread id that can write to this page when no other thread in the thread subtree is executing.

The entry *ParentThread[threadId]* indicates the parent thread of a thread. This table is used to check if a page being read by a thread is an MTROM page allocated by one of the parents of this thread.

The entry *ActiveThreads[threadId]* indicates the number of currently active threads in a subtree of threads rooted at node for $threadId$. When a thread in a subtree is created, the entries for all its ancestor threads are incremented. Similarly, when it terminates, the entries of

all its parent threads are decremented.

The two functions *READ_MTROM_OK* and *WRITE_MTROM_OK* as described below can be implemented either in software or hardware. The hardware implementation can be significantly faster when the tree of threads is not very deep. *thread_id* is the ID of the thread that executes this code. Also, $page\_num$ is always available in hardware as the most significant bits of the virtual address. It is assumed that the parent thread of the root thread is 0, but it could be any fixed number in the system, though 0 seems convenient and intuitive.

The function *READ_MTROM_OK* checks if the current thread is the owner of the MTROM page, or if it is a descendant of the owner of the page. If so, it returns true. Otherwise, it returns false, in which case the program should instrument the access and record the value change. Optionally, it can page fault or issue race warning, depending on whether the program is being run in test mode or in warning mode. This could be handled by a signal handler in software, as this is an unlikely case.

Similarly, the function *WRITE_MTROM_OK* checks if it is correct for the thread to write to that address. In either case, if the address is not MTROM address, the functions return false, in which case the access is instrumented.

When the compiler cannot statically rule out instrumenting a memory accesses, it uses the following technique to instrument the load or store instruction. If the address can never point to MTROM memory, and it needs instrumentation, then the compiler simply calls the load or store access instrumentation function. Otherwise, it inserts an MTROM access check as shown below.

Algorithm 5 Determining MTROM memory

```
Function READ_MTROM_OK
  owner = PageOwner[page_num];
  thread_id = current_thread_id;
  if(owner)
    while(thread_id and thread_id != owner_thread_id )
      thread_id = Parent[thread_id];

    if(thread_id)  return true;
    if(Test Mode) Page Fault;
    if(Warn Mode) Issue race warning
    return false;

  else
    return false

Function WRITE_MTROM_OK
  owner = Owner[page_num];
  if (owner)
    if(owner == current_thread_id and active_threads[owner] <= 1)
      return true;
    if(Test Mode) Page Fault;
    if(Warn Mode) Issue race warning
    return false;

  else
    return false;
```

Read Access Instrumentation:

```
if(READ_MTROM_OK(addr))
  uninstrumented load instruction
else
  call instrumented load
```

Write Access instrumentation:

```
if(WRITE_MTROM_OK(addr))
  uninstrumented write instruction
else
  call instrumented write
```

We caution that the techniques and algorithms discussed in this chapter are merely preliminary ideas.

Bibliography

[1] Martin Abadi, Cormac Flanagan, and Stephen N Freund. Types for safe locking: Static race detection for java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006.

[2] Sarita V Adve and Hans-J Boehm. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, 2010.

[3] Yehuda Afek, Guy Korland, and Arie Zilberstein. Lowering stm overhead with static analysis. In *Languages and Compilers for Parallel Computing*, pages 31–45. Springer, 2011.

[4] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K Shyamasundar. May-happen-in-parallel analysis of x10 programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 183–193. ACM, 2007.

[5] Jonathan Aldrich, Craig Chambers, EminGun Sirer, and Susan Eggers. Static analyses for eliminating unnecessary synchronization from java programs. In Agostino Cortesi

and Gilberto Fil, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*. 1999.

[6] Stephen Alstrup, Dov Harel, Peter W Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132, 1999.

[7] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[8] Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Software Engineering Conference, 2001. Proceedings. 2001 Australian*, pages 68–75. IEEE, 2001.

[9] Gross Axel. Evaluation of dynamic points-to analysis, 2004.

[10] Nels E. Beckman, Yoon Phil Kim, Sven Stork, and Jonathan Aldrich. Reducing stm overhead with access permissions. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, IWACO '09, 2009.

[11] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, 2010.

[12] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.

[13] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. *ACM SIGPLAN Notices*, 38(5):103–114, 2003.

[14] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[15] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, New York, NY, USA, 2008. ACM.

[16] Bruno Blanchet. Escape analysis for object-oriented languages: Application to java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, 1999.

[17] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, 1995.

[18] Hans-J Boehm and Sarita V Adve. Foundations of the c++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008.

[19] Michael D Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. Octet: Capturing and controlling

cross-thread dependences efficiently. In *ACM SIGPLAN Notices*, volume 48, pages 693–712. ACM, 2013.

[20] Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24:293–318, 1992.

[21] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Languages and Compilers for Parallel Computing*, pages 234–250. Springer, 1995.

[22] David Callahan. *The program summary graph and flow-sensitive interprocedual data flow analysis*, volume 23. ACM, 1988.

[23] Calin Cascaval, Colin Blundell, Maged Michael, Harold W Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: why is it only a research toy? *Communications of the ACM*, 51(11):40–46, 2008.

[24] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? In *ACM Queue - The Concurrency Problem, Volume 6 Issue 5*, pages 46–58, September 1 2008.

[25] Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM, 2007.

[26] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245. ACM, 1993.

[27] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, 1999.

[28] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. pages 258–269. ACM Press, 2002.

[29] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4:1–10, 2001.

[30] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[31] Madan Das, Gabriel Southern, and Jose Renau. Reducing logging overhead for deterministic execution. In *4th Workshop on Determinism and Correctness in Parallel Programming*, March 17, 2013.

[32] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 85–96. ACM, 2009.

[33] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Distributed Computing*, pages 194–208. Springer, 2006.

[34] Stephen A Edwards and Olivier Tardieu. Shim: A deterministic model for heterogeneous embedded systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(8):854–867, 2006.

[35] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J Boehm. Ifrit: Interference-free regions for dynamic data-race detection. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012.

[36] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, 2003.

[37] Xing Fang, Jaejin Lee, and Samuel P Midkiff. Automatic fence insertion for shared memory multiprocessing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 285–294. ACM, 2003.

[38] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN*

Symposium on Principles and practice of parallel programming, pages 237–246. ACM, 2008.

[39] Cormac Flanagan and Stephen N Freund. Type-based race detection for java. In *ACM SIGPLAN Notices*, volume 35, pages 219–232. ACM, 2000.

[40] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *ACM SIGPLAN Notices*, 39(1):256–267, 2004.

[41] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, 2009.

[42] Cormac Flanagan and Stephen N. Freund. Redcard: Redundant check elimination for dynamic race detectors. In *European Conference on Object Oriented Programming*, July 1-5, 2013.

[43] Cormac Flanagan, Stephen N Freund, and Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN Notices*, volume 43, pages 293–303. ACM, 2008.

[44] Cormac Flanagan and StephenN. Freund. Redcard: Redundant check elimination for dynamic race detectors. In *ECOOP 2013 Object-Oriented Programming*, volume 7920, pages 255–280. 2013.

[45] Cormac Flanagan and K Rustan M Leino. Houdini, an annotation assistant for esc/java.

In *FME 2001: Formal Methods for Increasing Software Productivity*, pages 500–517. Springer, 2001.

[46] Loukas Georgiadis and Robert E Tarjan. Finding dominators revisited. In *Proceedings of the fifteenth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 869–878. Society for Industrial and Applied Mathematics, 2004.

[47] Per Hammarlund, Rajesh Kumar, Randy B Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, et al. Haswell: The fourth-generation intel core processor. *IEEE Micro*, (2):6–20, 2014.

[48] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *ACM SIGPLAN Notices*, volume 38, pages 388–402. ACM, 2003.

[49] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60. ACM, 2005.

[50] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[51] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual Symposium on Principles of Distributed Computing*, pages 92–101. ACM, 2003.

[52] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[53] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM, 2001.

[54] Ali Jannesari and Walter F Tichy. Identifying ad-hoc synchronization for enhanced race detection. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[55] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *ACM SigPlan Notices*, volume 29, pages 171–185. ACM, 1994.

[56] John B Kam and Jeffrey D Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976.

[57] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

[58] Tom Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112. ACM, 1986.

[59] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory

fences. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 111–119. IEEE, 2010.

[60] Monica Lam, Ravi Sethi, JD Ullman, and AV Aho. Compilers: Principles, techniques, and tools, 2006.

[61] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, Jul 1978.

[62] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *In Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, 1993.

[63] Chris Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005.

[64] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[65] Chris Lattner and Vikram Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, Chigago, Illinois, June 2005.

[66] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in

a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[67] Lin Li and Clark Verbrugge. A practical mhp information analysis for concurrent java programs. In *of Lecture Notes in Computer Science*, pages 194–208. Springer, 2004.

[68] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[69] Virendra J Marathe, Michael F Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N Scherer III, and Michael L Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[70] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *ACM Sigplan Notices*, volume 44, pages 134–143. ACM, 2009.

[71] Alexander Matveev and Nir Shavit. Reduced hardware transactions: a new approach to hybrid transactional memory. In *Proceedings of the twenty-fifth annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 11–22. ACM, 2013.

[72] John Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *ACM SIGPLAN Notices*, volume 28, pages 129–139. ACM, 1993.

[73] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *In POPL*, pages 327–338. ACM Press, 2007.

[74] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *In PLDI*, pages 308–319. ACM Press, 2006.

[75] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. *ACM SIGPLAN Notices*, 38(10):167–178, 2003.

[76] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44(3):97–108, Mar. 2009.

[77] M Pnueli. Two approaches to interprocedural data flow analysis. *Program flow analysis: Theory and applications*, pages 189–234, 1981.

[78] Lori L Pollock and Mary Lou Soffa. An incremental version of iterative data flow analysis. *Software Engineering, IEEE Transactions on*, 15(12):1537–1549, 1989.

[79] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *ACM SIGPLAN Notices*, volume 38, pages 179–190. ACM, 2003.

[80] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *In PPoPP 03: Proceedings of the ninth ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 179–190. ACM Press, 2003.

[81] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Practical static race detection for c, 2011.

[82] Reese T Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138. ACM, 1959.

[83] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In *Runtime Verification*, pages 368–383. Springer, 2010.

[84] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 531–542. ACM, 2012.

[85] Vishwanath Raman. Pointer analysis–a survey. Technical report, Citeseer, 2004.

[86] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. IEEE, 2007.

[87] Martin C Rinard and Monica S Lam. The design, implementation, and evaluation of jade. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):483–545, 1998.

[88] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[89] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with llvm compiler. In *Runtime Verification*, pages 110–114. Springer, 2012.

[90] Julian Seward, Nicholas Nethercote, and Josef Weidendorfer. *Valgrind 3.3-Advanced Debugging and Profiling for GNU/Linux applications*. Network Theory Ltd., 2008.

[91] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213. ACM, 1995.

[92] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Annual Symposium on Principles of Programming Languages: Proceedings of the 23 rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 21, pages 32–41, 1996.

[93] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[94] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *Proceedings of the 2008 international Symposium on Software Testing and Analysis*, pages 143–154. ACM, 2008.

[95] Christoph von Praun and Thomas R Gross. Object race detection. In *ACM SIGPLAN Notices*, volume 36, pages 70–82. ACM, 2001.

[96] Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 115–128. ACM, 2003.

[97] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 205–214. ACM, 2007.

[98] John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Notices*, volume 39, pages 131–144. ACM, 2004.

[99] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36. ACM, 1995.

[100] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *OSDI*, volume 10, pages 163–176, 2010.

[101] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race con-

ditions via adaptive tracking. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 221–234. ACM, 2005.

[102] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '96, pages 81–92. ACM, 1996.

[103] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 121–132. IEEE, 2007.