ENHANCED DEBUGGING FOR DATA RACES IN PARALLEL PROGRAMS USING OPENMP

A Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Marcus W. Hervey
December 2016
ENHANCED DEBUGGING FOR DATA RACES IN PARALLEL PROGRAMS USING OPENMP

Marcus W. Hervey

APPROVED:

Dr. Edgar Gabriel, Chairman
Dept. of Computer Science

Dr. Shishir Shah
Dept. of Computer Science

Dr. Barbara Chapman
Dept. of Computer Science, Stony Brook University

Dean, College of Natural Sciences and Mathematics
Acknowledgements

A special thank you to Dr. Barbara Chapman, Ph.D., Dr. Edgar Gabriel, Ph.D., Dr. Shishir Shah, Ph.D., Dr. Lei Huang, Ph.D., Dr. Chunhua Liao, Ph.D., Dr. Laksano Adhianto, Ph.D., and Dr. Oscar Hernandez, Ph.D. for their guidance and support throughout this endeavor. I would also like to thank Van Bui, Deepak Eachempati, James LaGrone, and Cody Addison for their friendship and teamwork, without which this endeavor would have never been accomplished.

I dedicate this thesis to my parents (Billy and Olivia Hervey) who have always challenged me to be my best, and to my wife and kids for their love and sacrifice throughout this process.

"Our greatest weakness lies in giving up. The most certain way to succeed is always to try just one more time." – Thomas Alva Edison
ENHANCED DEBUGGING FOR DATA RACES IN PARALLEL PROGRAMS USING OPENMP

An Abstract of a Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Marcus W. Hervey
December 2016
Abstract

Parallel computing is pervasive. The variety and number of parallel hardware architectures increase daily. As this technology evolves, parallel developers will need productivity enhancing tools to analyze, debug and tune their parallel applications.

Current debugging tools that excel for sequential programs do not have the features necessary to help locate errors common to parallel programs (i.e. deadlock, livelock, priority inversion, race conditions). Data races, one type of race condition, are the most common software fault in shared-memory programs, and can be difficult to find because they are nondeterministic. Current data race detection tools are often plagued by numerous false positives and may exhibit inefficient execution times and memory loads. Enhancing the accuracy and efficiency of data race detectors can both increase the acceptance of shared memory programming models like OpenMP and improve developer productivity.

I describe a complementary analysis technique to detect data races in parallel programs using the OpenMP programming model. This hybrid analysis technique capitalizes on the strengths of stand-alone static and dynamic data race detection methods.

A proposed, future tool would implement the static analysis capabilities of the OpenUH compiler to complement the dynamic analysis techniques used in Sun (now Oracle) Thread Analyzer. This combination is expected to result in an improved debugging experience for developers intending to resolve data race errors resulting from poor programmer discipline, erroneous synchronization or inappropriate data scoping.
# Contents

1 Introduction 1

1.1 Motivation ......................................................... 2

1.2 Software Quality .................................................. 2

1.2.1 Failures, Faults, Errors (Defects) ........................... 3

1.2.2 Product Quality ............................................... 4

1.2.3 Process Quality ................................................ 5

1.2.4 Parallel Software Engineering – Software Quality Assurance ........................................... 5

1.3 Contributions ....................................................... 7

1.4 Thesis Organization ............................................... 8

2 Background 9

2.1 Shared-Memory Parallel Programming ............................ 10

2.2 Data Dependence ................................................... 11

2.2.1 True (Flow) Dependence ...................................... 11

2.2.2 Anti-dependence ............................................... 12

2.2.3 Output Dependence ............................................ 12

2.3 Synchronization ..................................................... 13

2.3.1 Synchronization Problems ..................................... 14

2.3.2 Synchronization Errors ......................................... 16

2.3.3 Synchronization Mechanisms .................................. 18
2.4 Synchronization Mechanisms in OpenMP

2.4.1 Mutual Exclusion Synchronization

2.4.2 Event Synchronization

2.4.3 Custom Synchronization

2.5 Overview of Open64 and OpenUH

3 Race Conditions and Data Races

3.1 General Races

3.2 Data Races

3.3 Formal Model

3.3.1 Feasible data race

3.3.2 Apparent data race

3.3.3 Actual data race

3.4 Benign Races

3.5 Causes of Data Races using OpenMP

3.5.1 Errors in Incorrect Parallelization

3.5.2 Errors in Data Scoping

3.5.3 Errors in Mutual Exclusion Synchronization

3.5.4 Errors in Event Synchronization

3.5.5 Errors in Custom Synchronization

4 Data Race Prevention and Detection

4.1 Ahead-of-Time

4.1.1 Static

4.2 On-the-Fly

4.2.1 Happens-Before Analysis

4.2.2 Lockset Analysis
4.3 Post-Mortem
   4.3.1 Flowback Analysis
   4.3.2 Replay Analysis
4.4 Model Checking
4.5 Transactional Memory
4.6 Comparison of Static Analysis and Dynamic Analysis Techniques

5 Current Data Race Detection Tools
   5.1 Static Analysis Tools
      5.1.1 LockLint
      5.1.2 Helgrind
      5.1.3 RacerX
   5.2 Dynamic Analysis Tools
      5.2.1 Thread Analyzer
      5.2.2 Thread Checker
      5.2.3 MultiRace
      5.2.4 RaceTrack
   5.3 Model Checkers

6 Challenges of Data Race Detection
   6.1 Debugging
      6.1.1 Statistical Errors
   6.2 Benign Data Races
   6.3 Limitations of Current Data Race Detection Tools

7 Dynamic Analysis Tool Experiments
   7.1 Standard Performance Evaluation Corporation OMP (SPEComp) Benchmarks
List of Figures

1.1 Quality Parallel Software ........................................... 5
2.1 Parallel Architectures ..................................................... 10
2.2 True Dependence .......................................................... 11
2.3 Anti-dependence ........................................................... 12
2.4 Output Dependence ......................................................... 13
2.5 Open64 Compiler ............................................................ 24
3.1 Classes of Race Conditions ............................................... 28
7.1 Collection Measurement Comparison on 3 Architectures ............ 59
7.2 Thread Checker Execution on Gomez .................................. 62
7.3 Thread Analyzer Execution on Kodos .................................. 63
8.1 Complementary Analysis Schematic with OpenUH and Thread Analyzer ................................................... 67
8.2 OpenMP Worksharing Constructs in PCFG [56] ...................... 70
8.3 PCFG of an OpenMP Example .......................................... 72
8.4 PCFG and PSSA of an OpenMP Example ............................. 73
List of Tables

2.1 Mechanisms for Synchronization Problems .......................... 18
4.1 Comparison of Static vs. Dynamic Analysis Techniques ............... 44
5.1 Data Race Detection Tools ............................................. 46
7.1 Parallel Architectures Experimented .................................. 56
7.2 Overview of the SPEComp Benchmarks [5] [90] ......................... 57
7.3 SPEComp Dynamic Analysis Collection Measurements .................. 58
7.4 Overview of the NPB [7] ............................................... 60
7.5 NPB Execution and Dynamic Analysis Collection Measurements ........ 61
Chapter 1

Introduction

The importance of high-quality parallel applications is increasing with the advent of inexpensive multicore computers entering the mainstream market. Computer hardware technology advances enable programmers to move beyond just executing software applications concurrently to now executing them in parallel.

Today, multicore computers are found everywhere, with manycore computers on the horizon. They exist in many forms, from gaming devices, embedded systems, desktop/laptop computers and even supercomputing. The presence of this evolving technology brings many opportunities as well as challenges.
1.1 Motivation

Many sequential applications can realize performance benefits by simply running on faster von Neumann architecture hardware. This is not the case for applications executing on parallel architecture hardware, as they may require extensive code modifications and performance tuning to exploit architecture improvements.

The value of parallel programming lies in its ability to realize enhanced performance speedup and efficiency. Despite its benefits, it can be equally challenging to maintain correctness and not introduce common parallel programming errors. Many mistakes result from the difficulty of conceptualizing programs executing in parallel and the lack of parallel productivity enhancing tools. As a result, parallel programs are frequently plagued with synchronization errors that create nondeterministic behavior and are extremely difficult to detect with standard sequential debugging techniques.

1.2 Software Quality

The success of many systems are significantly dependent on the quality of its software. The delivery of software quality is enabled by the implementation of proven software development processes through testing and effective debugging. The two major components of software quality include product and process quality.
Software quality can be defined in its simplest terms as: (1) software that conforms to its specification [Crosby] and/or (2) software that meets stakeholder’s expectations [Juran]. Debugging tools can detect conformance to standards like languages (Ada, C, C++, Fortran, Java) and APIs (MPI, OpenMP, POSIX). If we assume specifications effectively document stakeholder’s expectations, then we can further define software quality by its level of correctness and/or performance.

1.2.1 Failures, Faults, Errors(Defects)

One aspect of quality in software products is the absence of defects, where zero defects is the goal. A distinction between mistakes, defects, faults, errors and failures is necessary to understand how software product quality can be assessed. Mistakes can be caused by omission or commission. A defect occurs when software is not developed in accordance with its specification or expectation. A fault, is a type of defect, and is the root cause of a problem that causes software not to perform in an anticipated manner. Debugging can help to fix syntax/logic faults. An error is the manifestation of an undesirable feature of the software that is caused by a fault. Error handling and fault tolerance are techniques used to detect errors to prevent failures. Finally, a failure occurs when a software fault causes an error that results in a component or system to no longer function. The inappropriate use of synchronization in a parallel program is a mistake that results in a fault,
where the manifestation of this fault may lead to a data race error, resulting in a system’s failure. [6][58]

Mistake → Fault(Defect) → Error → Failure

The objective of fault tolerance is to preserve the delivery of correct service in the presence of active faults. Effective fault tolerance requires mechanisms to allow the failure to be detected, diagnosed, and if possible, corrected.

1.2.2 Product Quality

Desired software systems are those that are built and delivered to be resilient against errors and failures. Debugging and testing are essential in identifying software faults and preventing errors from occurring. Debugging assists in finding software faults, where testing is used to perform verification and validation. The successful identification of faults can also support the related goal of software fault tolerance where potential failures are prevented through the delivery of continued service in the presence of active faults. The earlier software faults are identified in the development process, the faster a quality software product can be delivered.
1.2.3 Process Quality

Edward Deming stated “... the quality of a product is directly related to the quality of the process used to create it.” Repeatable processes, based on sound software engineering principles, contribute to the development of quality software. Effective software engineering enables the generation of software with predictable cost, schedule and performance. This results in software applications that are portable, easy to program and maintain, highly scalable, and achieve the best performance, while minimizing software defects.

1.2.4 Parallel Software Engineering – Software Quality Assurance

Software engineering for parallel applications is essential to achieve correctness and performance objectives. Parallel software quality can be enabled by the use of effective software engineering principles in coordination with the tight coupling of people, process and technology as shown in Figure 1.1.

![Figure 1.1: Quality Parallel Software](image)
A repeatable software development process tailored for an organization’s culture is essential to quality software development. Once this process is defined, tools should be identified and used to automate and support the process. The people involved in the development must be trained and proficient on using the process and tools. Each of these components must be mature to meet the challenge of consistently delivering high quality parallel applications. My research is focused on improving debugging tools to improving parallel developer productivity.

Software engineering helps to ensure the quality of parallel software through the delivery of reliable, dependable and secure applications. Software Quality Assurance (SQA) activities include defect prevention, defect removal, and defect containment. Four techniques to ensure software quality include fault (defect) prevention, fault tolerance, fault removal and fault forecasting. Defect prevention can be achieved through walkthroughs, inspections, audits and review. Defect reduction can be achieved through testing. Defect containment can be achieved through fault tolerance and error handling.

Specific software engineering techniques include debugging, error handling, validation and verification. Debugging finds mistakes to prevent faults (syntax, semantic, logic). Verification occurs concurrently with software development and is the process of evaluating a systems or component to determine if it is being built right. Validation occurs after the software is developed and determines if the right product was built.
1.3 Contributions

A significant challenge for shared memory programming using OpenMP is the detection and prevention of data races. One programming problem for developers is appropriately synchronizing and scoping shared-memory variables to ensure program correctness. As a result, developers commonly generate data race errors that may not be detected using standard verification techniques, due to their non-deterministic nature.

Stand-alone static and dynamic tools are currently the most common approach to detecting data races, however, these techniques suffer from excessive detection execution times, lack precision and do not scale well. Combining the benefits of static and dynamic analysis in a complementary fashion, can improve the accuracy and practicality of data race debugging tools.

The contribution of this research is the identification of enhanced static analysis techniques used in the OpenUH compiler to reduce the amount of instrumentation needed by dynamic analysis tools, such as Thread Analyzer. This complementary approach should result in a more useful tool for developers, by enabling them to resolve data race defects faster and increasing their productivity. This research benefits developers of both shared-memory High Performance Computing (HPC) systems and mainstream multicore systems.
1.4 Thesis Organization

The work performed will be presented in the following chapters. Chapter 2 details the background relating to data dependence, synchronization, the OpenMP programming model and OpenUH compiler. Chapter 3 describes race conditions and data races. Chapter 4 discusses the challenges of data race detection. Chapter 5 details the current data race detection tools. Chapter 6 details the challenges of data race detection. Chapter 7 describes the dynamic analysis tool experiments. Chapter 8 discusses the proposed design and future implementation of RaceFree. Finally, Chapter 9 concludes the worked performed and provides ideas for future work.
But: program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

Chapter 2

Background

The goal of parallel programming in multicore machines is to effectively exploit its resources and achieve performance speedup and efficiency, while maintaining correctness. Two approaches to parallel programming include distributed memory and shared memory techniques. The shared-memory programming model is very attractive for multicore machines because it allows threads that share memory to easily communicate. The ways shared-memory parallel programs can be created include the use of language extensions (e.g. OpenMP for C, C++ and Fortran), parallel programming languages (e.g. Chapel, Fortress, X10), autoparallelization compilers (e.g. Open64/OpenUH, GCC, Intel, Sun) and parallel programming libraries (e.g. POSIX, Thread Building Blocks). Despite its ease of use, correctness for shared memory programs if actions performed by different
threads are not ordered in the right way can be difficult to attain. This chapter introduces some background material on data dependence and synchronization necessary to understand the challenges of shared-memory parallel program development.

2.1 Shared-Memory Parallel Programming

The advent of inexpensive parallel computing architectures (i.e. Field Programmable Gate Arrays (FPGAs), General Purpose Graphic Processing Units (GPGPUs), Accelerators, Cell processor-based systems, Sicortex systems) are leading the way for increased interest in parallel applications. These architectures can be represented in the Figure 2.1.

![Figure 2.1: Parallel Architectures](image)

The benefits of parallel programming is to achieve speedup and efficiency in performance, while maintaining the correctness by preventing data races and deadlocks.
2.2 Data Dependence

The identification of data dependences in sequential code is a major step in parallel program development. A data dependence in sequential code exists between two statements if they read from or write to a shared memory address in such a way that the order of execution must be maintained. The parallel version must respect these dependences.

There are three types of dependences that must be preserved for correctness in shared-memory programs: flow dependences, anti-dependences, output dependences.

2.2.1 True (Flow) Dependence

Flow (true) dependences, shown in Figure 2.2, occur when a definition (write) of a shared variable is followed by the use (read) of the same shared variable.

True Dependence

Statements S1, S2

S2 has a true dependence on S1 iff
S2 reads a value written by S1

S1: a = 1
S2: b = a

Figure 2.2: True Dependence
2.2.2 Anti-dependence

Anti-dependences, shown in Figure 2.3, occur when a use (read) of a shared variable is followed by the definition (write) of the same shared variable.

\[ S1: b = a \]
\[ S2: a = 1 \]

Figure 2.3: Anti-dependence

2.2.3 Output Dependence

Output dependences, shown in Figure 2.4, occur when a definition (write) of a shared variable is followed by another definition (write) of the same shared variable.
Output Dependence

Statements S1, S2.

S2 has an output dependence on S1
iff
S2 writes a variable written by S1

S1: a = 1
S2: a = 2

Figure 2.4: Output Dependence

2.3 Synchronization

Synchronization in parallel programs is needed to ensure the ordering relationships of work performed in different threads is maintained and to avoid data corruption. Previously, multithreaded programs used only lock-based approaches for synchronization. Today most multithreaded programs use a variety of different synchronization approaches.

Much of the effort associated with writing correct threaded programs is spent on synchronizing concurrent threads with respect to their data accesses or scheduling. To understand synchronization, it is important to discuss problems that require synchronization, errors that can result from poor synchronization and mechanisms that support synchronization.
2.3.1 Synchronization Problems

To understand the need for different synchronization mechanisms, it is important to understand the common problems that require synchronization in parallel programs. Popular synchronization problems for multithreaded programs include mutual exclusion, producer-consumer, readers and writers, and dining philosophers.

2.3.1.1 Critical Section

The critical section synchronization problem is the need to protect a shared variable from being accessed by multiple threads at the same time. Synchronization mechanisms are used to allow only one thread to access the shared variable at any instance. Once the thread with exclusive access is finished, other threads can access the shared variable. This synchronization mechanism prevents the variable from being overwritten and ensures the variable has the correct value. The length of critical sections must be minimized in order to maximize code performance.

2.3.1.2 Producers and Consumers

The producers and consumers synchronization problem occurs when access to a shared buffer needs to be managed. In this problem, producer threads write to the buffer, and consumer threads read from the buffer. Synchronization is needed to ensure that producer
threads and consumer threads do not access the buffer at the same time, producer threads
do not write to a full buffer, and consumer threads do not read from an empty buffer.

Two variations of the problem are the bounded buffer problem and unbounded buffer
problem. The size of the buffer is fixed in the bounded buffer problem. In the unbounded
buffer problem, the size of the buffer is not fixed.

2.3.1.3 Readers and Writers

The readers and writers synchronization problem occurs when there are multiple reader
and writer threads that compete for access to data. Multiple readers can access a shared
data object, however mutual exclusion is required when writers modify the shared data
object.

2.3.1.4 Dining Philosophers

The dining philosophers problem is a problem that manages the fairness of threads to gain
access to resources needed to continue thread execution. This problem is described as
a group of philosophers sitting at a table with a fork between each one, therefore each
philosopher requires two forks to eat, it is not possible for all philosophers to eat at the
same time. They must share forks in order to eat. The challenge of this problem is to
ensure that all of the philosophers get a chance to eat. Synchronizing the forks is needed
to ensure each philosopher has equal access to the forks.

2.3.1.5 Sleeping Barber

The sleeping barber synchronization problem is used to synchronize the actions within a client/server relationship. Specifically, it can be used to solve queuing problems. In this problem, the actions of the barber and the customers must be synchronized. When there are no customers, the barber sits in his chair and sleeps. As soon as a customer arrives, he either awakens the barber or, if the barber is cutting someone else’s hair, sits down in one of the vacant chairs. If all of the chairs are occupied, the newly arrived customer simply leaves.

2.3.2 Synchronization Errors

Software errors in parallel programs can be caused by the omission or misapplication of synchronization mechanisms. The four types of synchronization errors that can occur within shared-memory parallel programs include deadlocks/livelocks, priority inversions, and race conditions. Using synchronization in parallel programs is challenging applying too much synchronization can lead to deadlocks, where too little synchronization can lead to race conditions.
2.3.2.1 Deadlocks and Livelocks

A deadlock error occurs when the program hangs and is prevented from making progress towards the end state. A livelock error occurs when the program continues to execute, however does not make progress towards the end state. This condition is similar to an infinite loop.

2.3.2.2 Priority Inversion

A priority inversion error occurs when multiple threads are given a priority and a lack of synchronization causes a lower priority thread to preempt a higher priority thread.

2.3.2.3 Race Condition

A race condition error occurs when access to shared variables is not protected and enabling different threads to read and write to a shared variable. As a result, the value accessed by concurrent threads can obtain various values. Changes in input data or system load can cause nondeterminism with this error, making them hard to detect and debug.
2.3.3 Synchronization Mechanisms

Synchronization mechanisms are necessary to provide mutual exclusion or condition synchronization in parallel programs and are essential for program correctness. Several types of synchronization mechanisms have been provided in different languages to ensure data dependence and prevent synchronization errors. The primitive synchronizations include Mutex locks, condition variables and semaphores, where barriers and monitors represent more complex mechanisms. Table 2.1 list the mechanisms for synchronization problems.

<table>
<thead>
<tr>
<th></th>
<th>Locks/Barriers</th>
<th>Semaphores</th>
<th>Monitors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barriers</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Bounded Buffer</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Critical Section</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Dining Philosophers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Producers/Consumers</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Readers/Writers</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Sleeping Barber</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Table 2.1: Mechanisms for Synchronization Problems

2.3.3.1 Mutual Exclusion (Mutex) Locks

The most basic synchronization mechanism for an atomic action or mutual exclusion is that of a lock. Locks control the access of a thread to a region of code and specifically a shared variable. Mutex locks control access through the use of its two states: locked and unlocked.
2.3.3.2 Condition Variables

Condition variables are synchronization mechanisms that delay action until a specific condition becomes true. The indiscriminate use of locks can result in excessive overhead and can prevent idle threads. Condition variables are used with mutexes, and can allow more effective use of idle threads to perform work in the absence of the specified condition.

2.3.3.3 Reader and Writer Locks

Reader and Writer locks are used when a data structure is read often, but not written often. In this situation, multiple reads should not be blocked and can continue to proceed without causing any synchronization issues. Multiple threads can acquire a read lock which prevents any thread from writing to the shared data source. When a write lock is enforced all threads, except for the thread with the lock, must wait.

2.3.3.4 Monitors

Monitors are synchronization mechanisms that can be implemented as efficiently as semaphores, however are more structured and less prone to errors than semaphores [3]. To the programmer, monitors are only the data objects and the methods to access them. The compiler inserts code to lock or unlock the shared resource and eliminates the need for the programmer to explicitly lock and unlock, thereby making monitors safer to use and ensure
correctness.

### 2.3.3.5 Semaphores

Semaphores are synchronization mechanisms that allow for a more flexible approach to locking. Semaphores use a variable to control locks. Two different types of semaphores include binary semaphores and counting semaphores. Binary semaphores set or unset the variable, whereas counting semaphores can increment the variable without limit, but only decrement the variable to zero.

### 2.3.3.6 Spin Locks or Busy Waiting

Spinlocks are synchronization mechanisms that are held by one thread, but do not block other threads from performing other work until they control the lock. When a thread encounters a spinlock, it will repeatedly check the lock condition, or spin, until it succeeds in acquiring the lock.

### 2.3.3.7 Barriers

Barriers are synchronization mechanisms that forces all threads to wait until every thread has reached the barrier before allowing them to proceed. This allows all threads to be synchronized at the same place in the code.
2.4 Synchronization Mechanisms in OpenMP

Open Multi-Processing (OpenMP) is a shared-memory API that extends C, C++, and Fortran programs to provide fork-join parallelism. One strength of OpenMP is the ability to perform incremental parallelization of sequential programs. OpenMP consists of compiler directives, runtime library routines, environment variables and clauses [16].

One strength of OpenMP is the ability to perform incremental parallelization of sequential programs. While OpenMP can be used relatively easily, better performance and debugging tools are needed to enhance developer productivity.

OpenMP offers many constructs for the synchronization of threads. Specifically, OpenMP supports the fundamental synchronization needs of parallel programming including mutual exclusion synchronization and event/condition synchronization. OpenMP also provides the ability to construct custom synchronization [20].

2.4.1 Mutual Exclusion Synchronization

In OpenMP, mutual exclusion constructs are used to control access to a shared variable by providing a thread exclusive access to a shared variable for the duration of the construct. These constructs include the atomic directive, critical directive and runtime locks [20].

The Atomic construct is used to synchronize thread access to one statement within
a parallel region. The *Critical* construct is used to synchronize thread access to more than one statement in a parallel region. *Runtime Library Locks* enable a more flexible means for the programmer to implement mutual exclusion. This flexibility requires the programmer to be more disciplined in their use since locks are harder to apply and cause more synchronization errors.

### 2.4.2 Event Synchronization

Event synchronizations signal the occurrence of a specific event across multiple threads. The *Barrier* construct prevents any thread from proceeding past a point until all other threads reach that point. The *Ordered* construct is used with a For directive to direct the iterations of a loop to be executed in sequential order. The *Master* construct directs a region of code to only be executed by the master thread. The *Single* construct directs a region of code to be executed by only one thread.

### 2.4.3 Custom Synchronization

OpenMP does not provide constructs for every known synchronization mechanism, however it does provide constructs for the creation of custom synchronization for any problem. The *Flush* construct introduces thread-visible variables are written back to memory
to present a consistent view across all threads. It acts a fence to provides memory consistency. *Threadprivate* clause causes global variables to be local and persistent to a thread across multiple parallel regions.

### 2.5 Overview of Open64 and OpenUH

The design and implementation of a compiler is a substantial exercise in software engineering. A good compiler makes practical application of greedy algorithms (register allocation), heuristic search techniques (list scheduling), graph algorithms (dead-code elimination), dynamic programming (instruction selection), finite automata and push-down automata (scanning and parsing), and fixed-point algorithms (data-flow analysis). A compiler addresses problems such as dynamic allocation, synchronization, naming, locality, memory hierarchy management, and pipeline scheduling and makes many complex and diverse components work together.

The Open64 compiler is modularized, with different components that interact via a common Intermediate Representation (IR), called Winning Hierarchical Intermediate Representation Language (WHIRL). The major functional parts are the three front ends and the back end. The back end is composed of the global optimizer, loop nest optimizer, interprocedural analysis/optimizer, and code generator. There are also components for automatic parallelization and for generating source code from the IR. The WHIRL Optimizer
(WOPT) module performs data flow analysis at the procedure level and applies various SSA-based optimizations, such as copy propagation, dead code elimination, and partial redundancy elimination (SSAPRE). The WOPT is designed such that it can be invoked as a stand alone optimization phase, referred to as MAINOPT phase, or in conjunction with another compiler phase, referred to as PREOPT phases. In the latter mode, WOPT provides analysis and performs some basic optimizations to prepare the code for one of the other compiler components, e.g. the Loop Nest Optimizer (LNO).

Open64 uses five levels of WHIRL to allow more extensive analysis of code: Very High, High, Medium, Low, Very Low [93].

WHIRL consists of instructions and a symbol table. Each function/subroutine in a C, C++ or Fortran program is represented by a program unit (PU), and each PU is represented
by a WHIRL tree and a local symbol table. The WHIRL tree is composed of WHIRL nodes and the WHIRL symbol table is divided into the global part and the local part [94].

OpenUH is a modularized and robust open source compiler suite for OpenMP 2.5 in conjunction with C, C++, and Fortran 77/90/95 and the IA-64, x86, x86_64 Linux ABI and API standards.

The OpenUH compiler suite consists of OpenMP language extensions, the OpenUH compiler, Dragon analysis and performance tools. The optimizing compiler is based on the Open64 compiler, and is organized into two components: the front end and back end. The front end supports C, C++, and Fortran 77/90 programs. The back end consists of a global optimizer, a standalone inliner, an interprocedural analyzer (IPA), a loop nest optimizer (LNO), and a code generator. The back end also efficiently maps to Itanium, Opteron and x86 machines [69].
A race condition occurs in a parallel program when multiple threads compete (race) for access to a shared resource, and the access between the threads is not ordered. The presence of race conditions can cause programs to exhibit nondeterministic behavior. Some race conditions can generate incorrect results, while other race conditions are benign and can actually improve an application’s performance.

The efficient detection of software defects (faults) causing race condition errors remain a challenging problem in the field of computer science. Race condition errors are typically found in shared-memory parallel programs and can be initiated by a wide variety of software defects (faults), such as inappropriate synchronization. The unpredictable behavior of race conditions make them very difficult to detect and debug. In some situations,
they enable a program to generate correct results when a critical thread is accessed in the right order. In other situations, they can produce incorrect results when the order is not maintained. Testing for race condition errors can be unreliable and exhaustive testing is intractable [78].

Multiple names have been used in scientific literature to describe race conditions. Race conditions have been referred to as access anomalies, atomicity races, critical races, data races, determinancy races, general races, harmful shared-memory access, races, and synchronization race [78].

The variety of different descriptions for race condition errors led to Netzger’s research in creating a model to classify race conditions based on their detection tractability. He created the two categories of general races and data races to better understand detection limits and to assist in determining the most appropriate techniques for debugging.

General races introduce nondeterminism into deterministic programs, while data races are caused by the atomic failure of critical sections [78]. This distinction is helpful in determining the different debugging strategies required to identify and remedy a root cause defect of a race condition error. This model is also used to reason why general races found in programs designed to be deterministic require exhaustive race detection and programs designed to be nondeterministic with critical sections only requires partial race detection.

The classes of race conditions are show in Figure 3.1.
Netzer also introduced the concept of feasible, apparent and actual races to categorize data races \[78\]. Feasible races are more accurate for debugging, but are NP-hard to locate. Apparent races are less accurate for debugging, but can be identified more easily. Actual races are those that have occurred.

### 3.1 General Races

General (determinancy) races are a specific type of race condition error that occur in programs designed to be deterministic, but result in nondeterministic behavior due to unguaranteed synchronization access to a shared resource by multiple threads \[71\] \[78\]. This type of race condition can occur in both distributed memory and shared memory parallel programs. They also require more complex debugging techniques because the determination...
of the location of the software fault cannot be pinpointed within the program. Netzger determined that the precise location of general races in a parallel program is undecidable [78].

3.2 Data Races

Data (atomicity) races are a less complex form of race condition error that occur in nondeterministic programs where the atomicity of a critical section has failed [78]. More specifically, a data race can succinctly be described as an error in a shared-memory parallel program where the following conditions occur [13][100].

1. two or more threads access the same memory location concurrently

2. at least one of the accesses is a write

3. no synchronization mechanism is used to control the thread’s access to the shared memory location

3.3 Formal Model

Netzer’s research explored the properties of data races to create a formal model that characterized actual and potential behaviors exhibited by programs.
3.3.1 Feasible data race

Feasible general races and data races capture intuitive notions desired for debugging, however exactly locating feasible races is intractable (NP-hard). Those data races that could have occurred because of timing variations; necessary to eliminate all race condition errors, but impractical.

3.3.2 Apparent data race

Apparent data races are approximations of feasible races. The method of synchronization used in parallel programs determine the complexity of locating apparent races. Exhaustively locating all apparent data races is therefore always NP-hard. Therefore it is reasonable to detect only a subset of apparent data races for debugging purposes.

3.3.3 Actual data race

Actual data races are those data races that actually occurred. In this model, an actual data race indicates the possibility that the atomicity of a critical section may have actually failed. Actual data races can be easily located if the temporal ordering is known.
3.4 Benign Races

Benign races are intentionally designed to enhance the performance of nondeterministic programs. They are not harmful and their existence does not affect the correctness of the program. Benign data race conditions are commonly found in programs using lock-free, wait-free algorithms or double-checked locking[100].

There are some commonly used approaches that generate benign data races. First, programs using multiple threads reading, but not writing to a shared data location pose no risk of a data race. Second, programs where multiple threads write to a shared data location with the same value [20]. Commonly, the value written represents a true or false output. Third, programs that use multiple threads write to a shared data location, but a relaxed algorithm is used do not generate harmful data races either [20]. Finally, harmful data races may produce artifacts that exhibit race conditions, however sometimes these race conditions are also benign.

3.5 Causes of Data Races using OpenMP

There are many ways data races can be introduced into parallel programs. The use of patterns provide a simplified way to think about some of the common software mistakes that lead to data races. Specifying a complete set of patterns to detect data races is intractable,
however some common patterns can be identified. Such patterns are beneficial for use in debugging or the creation of test cases.

Suess [98] and Kosolov [64] have identified the most common mistakes generating potential race conditions. The mistakes they identified have led to our creation of common OpenMP data race mistake patterns that include incorrectly parallelizing code that is designed to run sequentially, misapplying mutual exclusion, misusing event and custom synchronization, and inappropriate data scoping.

### 3.5.1 Errors in Incorrect Parallelization

One strength of OpenMP is the ease of converting sequential code into parallel code. However, data races can be easily introduced into parallel programs if developers incorrectly parallelize inherently sequential code. These kinds of errors are commonly caused by the parallelization of loop-carried dependencies.

### 3.5.2 Errors in Data Scoping

Correct OpenMP programming depend on proper data scoping of variables. In OpenMP, variables can assume the data properties of shared, private or default. The incorrect scoping of variables in OpenMP programs can potentially lead to data races. Examples include not explicitly specifying a shared variable as private through the incorrect use of the
private, firstprivate or lastprivate clauses.

3.5.3 Errors in Mutual Exclusion Synchronization

Many parallel algorithms require mutually exclusive access to memory. In OpenMP, mutual exclusion is enabled by the use of the atomic directive, critical directive, reduction clause, taskgroup clause or runtime library locks. Missing or incorrect use of these directive, clauses and locks prevent the establishment of a critical region and can potentially cause data races.

3.5.3.1 Atomic directive

The atomic directive ensures mutually exclusive access to an atomic operation.

3.5.3.2 Critical directive

The critical directive creates mutual exclusion by enabling multiple threads to take turns accessing shared objects.
3.5.3.3 Reductions

The reduction clause provides a simplified and protected way to sum the values a shared variable within a loop. This approach creates mutual exclusive access to the shared variable that is summed.

3.5.3.4 Taskgroup directive

The taskgroup directive prevents a task from moving beyond a certain point until all of the child tasks and the descendents of the child tasks have reached that point.

3.5.3.5 Runtime library locks

Runtime library locks provide a more flexible approach to applying mutual exclusion to a shared resource. The flexibility provided by these locks also make it easier for developers to make mistakes leading to software faults that cause data races.

3.5.4 Errors in Event Synchronization

The ordering of access to shared resources by multiple threads is achieved by OpenMP’s event synchronization mechanisms. Errors can occur when one thread proceeds without waiting for another. In OpenMP, data races can occur by incorrectly applying `nowait`
clause, taskwait, master construct with lack of barrier, missing barrier construct or incorrect ordered construct.

3.5.4.1 Nowait directive

The nowait directive suppresses the barrier at the end of an associated construct.

3.5.4.2 Taskwait directive

The taskwait directive prevents a task from moving beyond a certain point until all tasks have reached that point.

3.5.4.3 Master directive

The master directive ensures a block of code is executed by the master thread only.

3.5.4.4 Barrier directive

The barrier directive prevents a thread from moving beyond a certain point until all threads have reached that point.
3.5.4.5 Ordered sections

The ordered directive enables a structured block to be executed in sequential order.

3.5.5 Errors in Custom Synchronization

In addition to the standard synchronization constructs provided by OpenMP, the API also provides mechanisms to create custom synchronization.

The flush directive ensures consistency of the variable between threads. A flush directive should be used before a shared variable is read to ensure that the value read is the value from memory and not a value held in a register. Similarly, a flush directive should be used after a shared variable is updated to ensure that the new value is written back to memory [39].
Correctness is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little.

---

Chapter 4

Data Race Prevention and Detection

Data races are the most common errors generated during shared-memory application development. The identification of faults leading to these errors remain a challenge, as current techniques generate results with poor accuracy and precision. The productivity of parallel developers can be enhanced through the advancement of data race prevention and detection mechanisms.

Research techniques to prevent data races include type-based techniques: parallel language development and transactional memory systems. Type-based approaches focus on providing thread safety mechanisms within the language. Specific examples of current research and implementations include rccjava, Cilk, X10, Chapel, Fortress, Cyclone, Concurrent Pascal, Extended Parameterized Race Free Java (EPRFJ). The strength in these
techniques are in the prevention of data race faults. Prevention of faults is easier than de-
tecting them, however the disadvantage of these techniques are the requirement for parallel
developers to learn new languages.

Debugging is the process of finding the specific software faults that cause programs to
behave in an undesirable way. A common way to debug sequential programs is through
the use of cyclic debugging techniques, that can be categorized into tracing and controlled
execution. In tracing, a program is instrumented and the trace is studied to find the fault
that caused the malfunction. In controlled execution, a program is instrumented to pause
its execution at breakpoints. A major problem with these debugging techniques is that
they can introduce heisenbugs \cite{73}. Cyclic debugging is not very effective for parallel
programs due to the non-determinism of data races.

Debugging for parallel programs must be able to detect fault despite non-determinism.
Much research has also been performed in various data race detection techniques. These
strategies can be organized into three categories: ahead-of-time (static or compile-time),
on-the-fly and post-mortem.
4.1 Ahead-of-Time

Ahead-of-Time analysis techniques are focused on performing data race detection before program execution. The strength of these techniques lie in its ability to analyze all execution paths of a program. The weakness of these techniques are they tend to be less accurate to the conservative nature of the compiler to guarantee correct interpretation.

4.1.1 Static

Static Analysis (compile-time) race detection techniques are usually the most sound and conservative, however result in more false positives. Control flow analysis examine the hierarchical flow of control between basic blocks within each procedure. Data flow analysis examines the global nature of data, including liveness and value of variables throughout a parallel program. Current static code analysis tools perform standards compliance, code coverage, requirements traceability verification and cyclomatic complexity. Previous tools and research based on static analysis techniques include RacerX \cite{33}, LockLint, -xvpara, SUNW_MP_WARN), Fluid, Prefix, ESP, Locksmith, Houdini/RCC.

Lei introduced a Parallel Data Flow Analysis (PDFA) framework that enables the OpenUH compiler to represent parallel programs \cite{56}. This framework provides the basis for further static data race detection improvements. The opportunity for improvement in using these techniques includes reducing the number of false positives by enhancing the
compiler to make intelligent decisions during analysis and enabling the compiler to be more aggressive and less conservative.

Current research areas look to enhance the usefulness of static data race detection through making improvements in non-concurrency (May-Happen-in-Parallel) analysis, interprocedural analysis, escape analysis, belief analysis and mutability analysis [71].

### 4.2 On-the-Fly

On-the-Fly or dynamic analysis techniques find software defects at run-time and are more precise than static techniques, because they do not generate false positives, however they heavily consume memory resources and be very time consuming to execute. Previous research on dynamic analysis techniques include lockset analysis, happens-before analysis, hybrid analysis and model checking. Current tools based on dynamic analysis techniques include Sun’s Thread Analyzer, Intel’s Thread Checker, RaceTrack [108] and MultiRace [82][83].

Dynamic race detectors can only find bugs in the execution paths that are actually taken at run-time. Dynamic analysis is costly with respect to execution time because every memory access is instrumented. Most dynamic analysis tools use either happens-before techniques or lockset-based techniques, however there have been some research in using a
hybrid approach which uses a combination of the two [79].

4.2.1 Happens-Before Analysis

Happens-before race detectors watch for thread accesses to a given piece of data that do not have any implied ordering between them. The advantage of happens-before analysis is that they do not give false positives. Current dynamic analysis tools that use happens-before analysis include Sun Thread Analyzer, Intel Thread Checker and Hewlett-Packard Visual Threads.

4.2.2 Lockset Analysis

In lockset analysis, the program runs on the principle of locking discipline with the assumption that all shared variables must be accessed within the protection of a lock and a specific lock should be used consistently when accessing that location in memory. Eraser is a dynamic race detection tool aimed at the lock-based synchronization and was the first dynamic tool to use this particular algorithm. It finds races by keeping track of locks held during program execution [91].
4.3 Post-Mortem

Post-Mortem race detection techniques are more precise than static based techniques, however are time consuming to execute and report as they require two phases: tracing and analysis. Previous research on post-mortem techniques include flowback analysis and replay analysis.

4.3.1 Flowback Analysis

Flowback analysis enables the examination of dynamic dependences in a parallel program’s execution without requiring the program to be re-executed, thereby reducing detection overhead time.

4.3.2 Replay Analysis

Replay analysis enables a reproducible scenario to be recorded in a replay log in a record/replay fashion. These tools allow the execution to be replayed twice for each detected data race enabling the order of each interleaving to be tried and tested for potential conflicting memory operations. If the same result is produced in each case, it is assumed that the data race is benign. Replay analysis reduces heisenbugs by using less intrusive methods by checking for data races during a replayed execution.
RecPlay [87] is a tool that combines execution replay with automatic on-the-fly data race detection with limited probe effect. It uses a happens-before based algorithm to perform the analysis.

4.4 Model Checking

Model checkers exhaustively tests for all execution paths and all possible thread interleavings. The length of time required for model checking analysis is typically unreasonable for use in an actual software development environment. Current model checkers include BLAST [14], SLAM [9], SPIN(PROMELA) [50], Keep It Simple and Sequential (Kiss) [85].

4.5 Transactional Memory

Transactional memory approaches enables aggressive optimizations to take place, but if a conflict is experienced, rolls back all optimizations to fix the conflict.
4.6 Comparison of Static Analysis and Dynamic Analysis Techniques

The accuracy and efficiency of data race detection can be improved by combining static analysis with dynamic analysis. Static and dynamic analysis data race detection techniques are well known, however, integrating the techniques to improve accuracy and precision remains a challenge. Table 4.1 lists the comparison of static and dynamic analysis techniques.

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
<th>Opportunities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>- regardless of input</td>
<td>- produces many false positives</td>
<td>Concurrency analysis</td>
</tr>
<tr>
<td></td>
<td>- regardless of execution path</td>
<td>- not scalable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- all flows through a program</td>
<td>- conservative, many false positives</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- benign, hard to distinguish</td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td></td>
<td>- costly wrt to execution time and memory</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- limited routes of analysis</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of Static vs. Dynamic Analysis Techniques
Controlling complexity is the essence of computer programming.

Brian Kernighan

Chapter 5

Current Data Race Detection Tools

Many research and commercial tools have been developed to identify data races in shared-memory programs. These tools directly relate to data race prevention/detection techniques categories previously introduced. The most popular static data race detection tools include Warlock, LockLint, Helgrind (based on the Eraser algorithm), and Locksmith. The most popular dynamic data race detection tools include Intel’s Thread Checker (based on Assure technology), Sun’s Thread Analyzer (previously Data Race Detection Tool), and Hewlett Packard’s Visual Threads. A list of past and present data race detection tools are shown in Table 5.1.
<table>
<thead>
<tr>
<th>Ahead-of-Time</th>
<th>On-the-Fly</th>
<th>Post-Mortem</th>
<th>Model Checking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>[48, Hanzinger, 2002]</td>
</tr>
<tr>
<td>(ESC) [28, Detlefs, 1998]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESC/Java [37, Flanagan, 2001]</td>
<td>Distributed Debugging Tool</td>
<td></td>
<td>KISS Technique [85, Qadeer, 2004]</td>
</tr>
<tr>
<td></td>
<td>(DDT) [2, Allinea, 2010]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Detection via</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalable Program Analysis</td>
<td>Eraser [91, Savage, 1997]</td>
<td></td>
<td>SLAM [9, Ball, 2002]</td>
</tr>
<tr>
<td>(ESP) [27, Das, 2002]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Detection [79, O’Callahan, 2003]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Liblit’s [70, Liblit, 2003]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Liblit’s [70, Liblit, 2003]</td>
<td>RaceStand [63, Kim, 2005],</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[62, Kim, 1999]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LockLint [99, Sun, 2006]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PREfix [17, Bush, 2000]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RacerX [33, Engler, 2003]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vpara/xvpara (scoping) [72, Lin, 2004]</td>
<td>Thread Checker [81, Petersen, 2003],</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[11, Banerjee, 2006]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Data Race Detection Tools
5.1 Static Analysis Tools

Static analysis tools find software defects at compile-time, however produce false positives. Examples of static analysis tools include LockLint, Helgrind, and RacerX.

5.1.1 LockLint

LockLint is a Pthreads raced detection tool based on Warlock [96] and Lint, the C program checker [59] that can detect the most common causes of race conditions and deadlocks. LockLint analyzes the use of mutex and multiple readers/single writer locks, and identifies for inconsistent use of these locking techniques. Semaphores and condition variables are not recognized as locks by LockLint.

5.1.2 Helgrind

Helgrind looks for memory locations which are accessed by more than one thread. Helgrind records which of the program’s (pthread_mutex) locks were held by the accessing thread at the time of the access [106].
5.1.3 RacerX

RacerX is a tool created by Dawson Engler and Ken Ashcroft from Stanford University, that uses flow-sensitive, interprocedural dataflow analysis to detect both race conditions and deadlocks. RacerX is an implementation that proposes to rank detected errors using specific criteria. It helps the user prioritize the results of the race detection analysis by ranking errors from most severe to least severe [33].

5.2 Dynamic Analysis Tools

The most popular commercial dynamic analysis tools are Thread Analyzer and Thread Checker.

5.2.1 Thread Analyzer

Sun Data Race Detection Tool (DRDT) (has been renamed the THREAD ANALYZER (THA)) is used to detect data-races that occur during the execution of a single, multi-threaded process.
5.2.2 Thread Checker

Intel Thread Checker detects potential deadlocks and data races for OpenMP and multithreaded codes. Thread Checker uses happens-before analysis for Win32 and POSIX multithreaded programs. Lockset-based analysis is used for OpenMP programs. Thread Checker performs data race analysis on-the-fly.

Thread Checker has two modes of operation: source instrumentation and binary instrumentation. Source instrumentation is enabled using both `-openmp -tcheck` flags, while binary instrumentation is enabled by just using the `-openmp` flag. Source instrumentation cannot interpret `omp_get_thread_num()` or any other command dependent on a specific thread or thread count, whereas binary instrumentation can.

5.2.3 MultiRace

MultiRace combines improved versions of Djit and Lockset - two very powerful on-the-fly algorithms for dynamic detection of apparent data races [82][83].

5.2.4 RaceTrack

RaceTrack is a dynamic race detection tool that tracks the actions of a program and reports a warning whenever a suspicious pattern of activity has been observed. RaceTrack uses a
novel hybrid detection algorithm and employs an adaptive approach that automatically di-
rects more effort to areas that are more suspicious, thus providing more accurate warnings
for much less overhead [108].

5.3 Model Checkers

Model checking is a formal technique that enables the verification of desired behavioral
properties for a system. Verification is performed by inspecting all of the states of a repre-
sentative model of the system.
Chapter 6

Challenges of Data Race Detection

Data races detection is a challenging problem for shared memory programs because they are nondeterministic. Therefore, it is difficult to guarantee that a race condition will not occur. Current static and dynamic race detection techniques generate many false positives that limit developer productivity. The typical stand alone approaches of static and dynamic analysis are not sufficient. This chapter describes the challenges with respect to data race detection in shared-memory parallel computing systems.
6.1 Debugging

Testing exposes errors, while debugging locates the faults that case the errors. Many refer
to debugging as an art, but I think a discipline can be applied to it to make it more of
a science. Debugging find faults made by mistakes that could create errors leading to
failures.

The implementation lifecycle is as follows:

edit → compile → test → debug

The reasons to perform debugging are to answer the following questions:

1. What was the originating cause of the error? (syntax, logic)

2. Where in the software lifecycle was the fault introduced? (specification, design,
   implementation)

3. Where was the error exposed? (compile, link, run-time (execution))

6.1.1 Statistical Errors

Data races are synchronization errors that are caused by faults in the software. These
errors can be classified statistically as Type I or Type II. To improve the performance of
debugging tools, these types of errors must be minimized.
6.1.1.1 Type I Errors

Type I errors (false positives) are the erroneous identification of a threat or dangerous condition that turns out to be harmless. A result that is erroneously positive when a situation is normal. Reports an error when an error does not exist.

6.1.1.2 Type II Errors

Type II error (false negatives) are the erroneous identification of a benign condition that turns out to be harmful. A result that appears negative but fails to reveal a situation. Fails to support an error that exists.

6.2 Benign Data Races

Differentiation between benign and harmful data races is needed to focus triaging activities to reduce the number of false positives generated \[77\].

Examples of benign data races include multiple reads and multiple writers that write the same value \[20\].
6.3 Limitations of Current Data Race Detection Tools

The limitations that exist for current dynamic analysis methods and tools include missing data races that are not in the runtime execution path and their dependence on varied input data and runtime environments [100][105].

The Thread Analyzer User Guide states that ”...experiments should be performed with varied factors such as different input data, a different number of threads, varied loop schedules or even different hardware” [100]. Moreover, dynamic analysis tools’ inability to scale (memory usage, execution time, eg.) make them unlikely to be used to debug large parallel applications.
Chapter 7

Dynamic Analysis Tool Experiments

This chapter describes the experiments performed during my research. At this stage of my work, the proposed static analysis tool has not been completed. Therefore, experiments were conducted to measure the amount of collection overhead introduced by dynamic analysis tools during the tracing process. The precision of the dynamic analysis tools in performing data race detection was not measured.

The SPEComp benchmarks and NAS Parallel Benchmarks were used to compare Thread Checker and Thread Analyzer collection duration measurements with the benchmark application execution measurements. The value of these benchmarks is that they allow parallel architectures and applications to be evaluated with real applications.

Performing experiments using the benchmarks and these dynamic analysis tools was a
learning experience. The first error I made was trying to execute the largest benchmarks classes. These experiments were extremely time consuming or were never completed. The second error I made was running the tools using source instrumentation instead of binary instrumentation. The use of source instrumentation resulted in many false positives and therefore was not realistic.

Three different parallel architectures (Kodos, Optimus, Gomez) with a variety of processing capability, memory and operating systems were used to perform the experiments. Kodos had the slowest processing ability and least memory, Gomez had the fastest processing ability and the most memory, and Optimus was in between. The specifications of each parallel architecture is shown in Table 7.1.

<table>
<thead>
<tr>
<th></th>
<th>Kodos</th>
<th>Optimus</th>
<th>Gomez</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>4 SPARC CPUs</td>
<td>Sun V40z Server 8 dual core AMD Opteron 880</td>
<td>4 IA-64 CPUs</td>
</tr>
<tr>
<td>Frequency</td>
<td>750 MHz</td>
<td>1GHz</td>
<td>1.5GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>7GB RAM</td>
<td>32 GB RAM</td>
<td>40 GB RAM</td>
</tr>
<tr>
<td>OS</td>
<td>Solaris 9 12/02</td>
<td>CentOS release 4.4</td>
<td>SUSE Server 9</td>
</tr>
</tbody>
</table>

Table 7.1: Parallel Architectures Experimented
7.1 Standard Performance Evaluation Corporation OMP (SPECmp) Benchmarks

SPECmp is a benchmark suite developed by the Standard Performance Evaluation Corporation (SPEC) and designed to evaluate performance based on OpenMP applications. SPECmp2001 was used for our experiments and includes a Medium and a Large data set. The Medium data set is targeted for shared-memory multiprocessors systems of about 10 CPUs, whereas the Large data set is targeted for systems with 30 CPUs or more [5]. The suite is composed of 12 different real world applications with different code and problem sizes as shown in Table 7.2.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Real World Application</th>
<th>Language</th>
<th>SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>Chemistry/biology</td>
<td>C</td>
<td>13500</td>
</tr>
<tr>
<td>applu</td>
<td>Fluid dynamics/physics</td>
<td>Fortran</td>
<td>4000</td>
</tr>
<tr>
<td>apsi</td>
<td>Air pollution</td>
<td>Fortran</td>
<td>7500</td>
</tr>
<tr>
<td>art</td>
<td>Image recognition/neural networks</td>
<td>C</td>
<td>1300</td>
</tr>
<tr>
<td>equake</td>
<td>Earthquake modeling</td>
<td>C</td>
<td>1500</td>
</tr>
<tr>
<td>facerec</td>
<td>Face recognition</td>
<td>Fortran</td>
<td>2400</td>
</tr>
<tr>
<td>fma3d</td>
<td>Crash simulation</td>
<td>Fortran</td>
<td>60000</td>
</tr>
<tr>
<td>gafort</td>
<td>Genetic algorithm</td>
<td>Fortran</td>
<td>1500</td>
</tr>
<tr>
<td>galgel</td>
<td>Fluid dynamics</td>
<td>Fortran</td>
<td>15300</td>
</tr>
<tr>
<td>mgrid</td>
<td>Multigrid solver</td>
<td>Fortran</td>
<td>500</td>
</tr>
<tr>
<td>swim</td>
<td>Shallow water modeling</td>
<td>Fortran</td>
<td>400</td>
</tr>
<tr>
<td>wupwise</td>
<td>Quantum chromodynamics</td>
<td>Fortran</td>
<td>2200</td>
</tr>
</tbody>
</table>

Table 7.2: Overview of the SPECmp Benchmarks [5] [90]
7.1.1 Experiments with SPEComp

Experiments were performed with SPEComp to assess the collection time needed by Thread Checker and Thread Analyzer to analyze the benchmark applications for parallel programming errors. Only the dynamic analysis tools collection measurements with these benchmark applications were conducted and are found in Table 7.3.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Optimus Thread Checker (binary mode)</th>
<th>Gomez Thread Checker (binary mode)</th>
<th>Kodos Thread Analyzer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp_m</td>
<td>4 threads 116.762</td>
<td></td>
<td></td>
</tr>
<tr>
<td>applu_m</td>
<td>297.221 57.045</td>
<td>143.965</td>
<td></td>
</tr>
<tr>
<td>apsi_m</td>
<td>4209.317 68.911</td>
<td>5116.664</td>
<td></td>
</tr>
<tr>
<td>art_m</td>
<td>84.625 8096.117</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equake_m</td>
<td>343.048 17.380</td>
<td>353.537</td>
<td></td>
</tr>
<tr>
<td>fma3d_m</td>
<td>47.032 42.859</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gafort_m</td>
<td>155.828 1047.175</td>
<td></td>
<td></td>
</tr>
<tr>
<td>galgel_m</td>
<td>275.227 15.495</td>
<td>314.387</td>
<td></td>
</tr>
<tr>
<td>mgrid_m</td>
<td>234.781 700.336</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3: SPEComp Dynamic Analysis Collection Measurements

7.1.2 SPEComp Comparison with Different Parallel Architectures

Figure 7.1 graphically displays a comparison of the results obtained from the experiments on the three different architectures. Thread Checker experiments with SPEComp were
performed on both the Optimus and Gomez systems, and Thread Analyzer experiments were performed on the Kodos system.

Figure 7.1: Collection Measurement Comparison on 3 Architectures

The experiment results consistently show that the slower system, Kodos, takes longer to perform the dynamic analysis with these tools, as expected. It is notable to see the *apsi* application took over 1 hour to complete for Thread Analyzer and Thread Checker, and the *art* application took over 2 hours to complete with Thread Analyzer. Reducing the dynamic analysis collection time for these tools would be beneficial for parallel development.
7.2 National Aeronautical and Space Administration (NASA) Numerical Aerodynamic Simulation (NAS) Parallel Benchmarks

The National Aeronautical and Space Administration (NASA) Numerical Aerodynamic Simulation (NAS) Parallel Benchmarks, shown in Table 7.4, is a popular benchmark suite that simulates 10 computational fluid dynamics applications, mostly a variety of Partial Differential Equation (PDE) solvers.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Real World Application</th>
<th>Language</th>
<th>SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Tridiagonal (BT)</td>
<td>Block tridiagonal PDE solver</td>
<td>Fortran</td>
<td>2591</td>
</tr>
<tr>
<td>Conjugate Gradient (CG)</td>
<td>System of equation solver</td>
<td>Fortran</td>
<td>640</td>
</tr>
<tr>
<td>Data Cube (DC)</td>
<td>Data cube operator</td>
<td>C</td>
<td>2872</td>
</tr>
<tr>
<td>Embarrassingly Parallel (EP)</td>
<td>Complex pair random number generator</td>
<td>Fortran</td>
<td>182</td>
</tr>
<tr>
<td>3-D FFT PDE (FT)</td>
<td>Fast Fourier Transform PDE solver</td>
<td>Fortran</td>
<td>838</td>
</tr>
<tr>
<td>Integer Sort (IS)</td>
<td>Integer bucket sort</td>
<td>C</td>
<td>567</td>
</tr>
<tr>
<td>Lower Upper (LU)</td>
<td>System of nonlinear PDE solver</td>
<td>Fortran</td>
<td>2656</td>
</tr>
<tr>
<td>Multigrid (MG)</td>
<td>3D Poisson equation solver</td>
<td>Fortran</td>
<td>909</td>
</tr>
<tr>
<td>Scalar Pentadiagonal (SP)</td>
<td>System of nonlinear PDE solver</td>
<td>Fortran</td>
<td>1908</td>
</tr>
<tr>
<td>Unstructured Adaptive (UA)</td>
<td>Heat equation solver</td>
<td>Fortran</td>
<td>5002</td>
</tr>
</tbody>
</table>

Table 7.4: Overview of the NPB [7]
7.2.1 Experiments with NPB

Experiments were performed with NPB to assess the collection time needed by Thread Checker and Thread Analyzer to analyze these applications for parallel programming errors. I performed measurements using Class S for small problem size and Class W for small-memory systems, where I experienced the most success experimenting with Class S and Class W using Thread Analyzer and Thread Checker. The results of the measurements collected are shown in Table 7.5.

<table>
<thead>
<tr>
<th>Application</th>
<th>Gomez</th>
<th>Kodos</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Execution</td>
<td>Collection</td>
</tr>
<tr>
<td>Block Tridiagonal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bt.S</td>
<td>0.11</td>
<td>14.30</td>
</tr>
<tr>
<td>bt.W</td>
<td>2.04</td>
<td>107.16</td>
</tr>
<tr>
<td>Conjugate Gradient</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cg.S</td>
<td>0.11</td>
<td>14.44</td>
</tr>
<tr>
<td>cg.W</td>
<td>0.52</td>
<td>19.25</td>
</tr>
<tr>
<td>Data Cube</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dc.S</td>
<td>0.003</td>
<td>13.63</td>
</tr>
<tr>
<td>dc.W</td>
<td>0.003</td>
<td>10.82</td>
</tr>
<tr>
<td>Embarrassingly Parallel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ep.S</td>
<td>1.43</td>
<td>18.61</td>
</tr>
<tr>
<td>ep.W</td>
<td>2.75</td>
<td>21.68</td>
</tr>
<tr>
<td>3-D FFT PDE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ft.S</td>
<td>0.21</td>
<td>19.83</td>
</tr>
<tr>
<td>ft.W</td>
<td>0.31</td>
<td>26.15</td>
</tr>
<tr>
<td>Integer Sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>is.S</td>
<td>0.11</td>
<td>13.44</td>
</tr>
<tr>
<td>is.W</td>
<td>0.31</td>
<td>12.44</td>
</tr>
<tr>
<td>Lower Upper</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lu.S</td>
<td>0.11</td>
<td>16.98</td>
</tr>
<tr>
<td>lu.W</td>
<td>5.99</td>
<td>667.73</td>
</tr>
<tr>
<td>Multigrid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mg.S</td>
<td>0.11</td>
<td>13.54</td>
</tr>
<tr>
<td>mg.W</td>
<td>0.82</td>
<td>32.45</td>
</tr>
<tr>
<td>Scalar Pentadiagonal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sp.S</td>
<td>0.11</td>
<td>15.36</td>
</tr>
<tr>
<td>sp.W</td>
<td>6.10</td>
<td>332.26</td>
</tr>
<tr>
<td>Unstructured Adaptive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ua.S</td>
<td>1.43</td>
<td>50.40</td>
</tr>
<tr>
<td>ua.W</td>
<td>7.52</td>
<td>229.18</td>
</tr>
</tbody>
</table>

Table 7.5: NPB Execution and Dynamic Analysis Collection Measurements
7.2.2 Thread Checker Results

Figure 7.2 graphically displays the application execution duration and Thread Checker collection duration for each of the NAS benchmarks experimented on Gomez.

![Thread Checker NPB Results on Gomez](image)

Figure 7.2: Thread Checker Execution on Gomez

The experiment results show BT, LU, SP, UA applications yielded Thread Checker collection duration results significantly longer the application execution durations.
7.2.3 Thread Analyzer Results

Figure 7.3 graphically displays the execution time and Thread Analyzer collection time for each of the NAS benchmarks experimented on Kodos.

![Thread Analyzer NPB Results on Kodos](image)

Figure 7.3: Thread Analyzer Execution on Kodos

The experiment results show \textit{BT, LU, SP, UA} applications yielded Thread Analyzer collection duration results significantly longer the application execution durations. The magnitude of the collection duration results for the same benchmark applications were similar to the results obtained on the other architectures using Thread Checker. As to be expected with Kodos being a slower system, the duration to collect on this architecture took longer than the other architectures, however it is notable that the \textit{LU, SP, UA}
applications took multiple hours to complete.

7.3 Overall Experiment Analysis

Analyzing the results obtained, it is evident a significant overhead is experienced using the dynamic analysis tools. Therefore, I conclude that any reduction achieved with static analysis in combination with dynamic analysis will result in a faster, better data race detection experience for parallel programmers.
First solve the problem. Then, write the code.

Chapter 8

RaceFree

The identification of data races in parallel programs is extremely difficult. The detection of feasible races and the exact determination of nonconcurrency, within parallel programs, are both NP-hard problems [78] [103].

Static and dynamic data race detection techniques have been well documented, with each technique having certain advantages and disadvantages [91] [96]. Static analysis techniques are faster, more conservative, and sound, while dynamic analysis techniques are more precise [4] [35]. Static analysis tools have a tendency to produce many false positives, whereas dynamic analysis tools take longer to execute and may not trace all possible paths through a program.
Complementary analysis is an approach to combine static and dynamic analysis techniques, by using the soundness of static analysis to symbiotically steer the precision of dynamic analysis. Complementary analysis is not a new concept and its benefits have been previously explored [4] [35] [95].

The focus of my research includes extending Choi’s [23], Huang’s [56], Lee’s [66] and Lin’s [71] work, to identify segments of OpenMP programs where data races cannot exist. These code segments are categorized as "race free". Maximizing the amount of "race free" code identified statically and eliminated from dynamic analysis should result in an improved data race debugging experience. A tool that can assist in the removal of data races would also enable other classical compiler optimizations and analysis techniques to be applied to parallel programs.

Dynamic analysis tools, like Thread Analyzer and Thread Checker, instrument every memory access, lengthening the execution time to detect data races in OpenMP programs. The aim of my research is to evolve static analysis techniques such that the number of variables monitored by dynamic analysis tools can be incrementally reduced to the minimum set required for instrumentation. This should result in a faster data race detection time using the complementary analysis approach.

RaceFree is a proposed tool, to be implemented in the OpenUH compiler, that will perform static analysis of OpenMP parallel programs. RaceFree will specify segments of
code that are “race free” using annotations that are recognized by Thread Analyzer. The annotated program will be provided to Thread Analyzer where the final data race set is generated. A schematic of this approach is shown in Figure 8.1.

Figure 8.1: Complementary Analysis Schematic with OpenUH and Thread Analyzer

8.1 Parallel Control Flow Graph (PCFG)

A Parallel Data Flow Analysis (PDFA) framework, within the OpenUH compiler, can be used to perform static analysis before OpenMP translation. Data flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A control flow graph is a representation of all paths that might be
traversed through a program during its execution. The PDFA uses the intermediate representations of a Parallel Control Flow Graph (PCFG) and Parallel Single Static Assignment (PSSA) to analyze OpenMP programs [26] [56] [66].

The design of the PCFG [56] is based on the Synchronized Control Flow Graph (SCFG) proposed by Callahan and Subhlok [19], the Program Execution Graph (PEG) proposed by Balsundaram and Kennedy [8], and the Partial Order Execution Graph (POEG) proposed by Dinning [30].

The PCFG is a directed graph \((N, E, s, e)\), where \(N\) is the set of nodes to include basic nodes, composite nodes, super nodes, and barrier nodes; \(E\) is a set of directed edges including sequential edges, parallel edges and conflict edges; \(s\) and \(e\) represent the entry and exit of a parallel region. The nodes and edges of the PCFG are defined as follows:

**Definition 1. Basic node:** a basic node is a basic block with one entry point, and one exit point; an \(\texttt{omp flush}\) directive is a special statement that synchronizes shared variables and is also considered to be a basic node.

**Definition 2. Composite node:** a composite node is composed of an OpenMP worksharing or synchronization construct and the basic nodes associated with it. It can also be consecutive basic nodes executed by all threads in a parallel region that are not enclosed in an OpenMP construct.
Definition 3. **Super node**: a super node contains all composite nodes between two barriers. Super nodes are separated by barriers, and are executed in the order of “happens before”.

Definition 4. **Barrier node**: a barrier node contains the `omp barrier` directive only. It is used to synchronize data and thread execution. A barrier node represents both an explicit barrier as well as an implicit barrier.

Definition 5. **Sequential edge**: a sequential edge indicates a control flow path executed by a thread or by all threads. The code inside an OpenMP construct is also connected via sequential edge since the code is executed by either one thread or all threads.

Definition 6. **Parallel edge**: a parallel edge is used to represent a branch executed by one or more threads, but not all threads. A parallel edge is only used at a point where there are multiple paths that different threads may take.

Definition 7. **Conflict edge**: a conflict edge indicates a situation where two different threads are accessing the same memory location and at least one of them is a write. Conflict edges are bidirectional and join basic blocks where a conflict of a shared variable exists.

The PCFG representation of the OpenMP worksharing constructs for `omp single`, `omp sections`, `omp for` and `omp critical` are shown in Figure 8.2.
8.1.1 Parallel Single Static Assignment

Static Single Assignment (SSA) is an intermediate representation where each variable is only assigned a value once and is defined before it is used. Once the PCFG is created, a modified SSA, called Parallel Static Single Assignment (PSSA) is used to help analyze the program. PSSA ensures all uses of a variable are reached by exactly one (static) assignment to the variable. The PSSA form has three confluence functions: \( \phi \) (the same
as the SSA $\phi$-function [26] and two additional functions: $\psi$ and $\pi$ [66]. These functions are useful in understanding how variables are affected throughout a program. They are defined as:

**Definition 8.** $\phi$-function: a $\phi$-function is placed at the join of control flow edges join [66].

**Definition 9.** $\psi$-function: a $\psi$-function is placed at the join of parallel control flow edges join [66].

**Definition 10.** $\pi$-function: a $\pi$-function is placed where there is the use of a shared variable at a conflict edge [66].

In PSSA, the $\phi$-functions are placed at the end of control flow graph joins where multiple values of the same variable are possible. $\psi$-functions are placed at the end of parallel regions where multiple values of the same variable are possible. $\pi$-functions are placed where multiple values of a shared variable may have different values from multiple threads, potentially causing a conflict.

### 8.1.2 An Example of the PCFG/PSSA and RaceFree algorithms

A program example is useful to visualize the PCFG and PSSA representations. Figure 8.3 displays an OpenMP program and its PCFG representation.
This example includes one super node with an entry and an exit node. The supernode consists of three composite nodes to represent the `omp single`, `omp for` and `omp for nowait` constructs. Parallel and sequential edges are also displayed to represent the control flow within this PCFG.

Once the PCFG is constructed, PSSA is used to perform analysis and optimizations. PSSA is created by first computing the execution order of blocks with synchronizations. Next $\phi$-functions are inserted at each branch to represent the possible definitions to reach...
that point. The $\psi$-functions are inserted at the end of each supernode to represent the possible definitions in different threads for a shared variable in the supernode. After the $\phi$-functions and $\psi$-functions are placed, variables are then renamed to ensure single assignment. Finally, the conflict edges of the PCFG are used to determine where $\pi$-functions are placed and to represent the possible conflicts of a variable from multiple threads. Fig. 8.4 displays the PCFG and PSSA for the example in Fig. 8.3.

![Figure 8.4: PCFG and PSSA of an OpenMP Example](image-url)
8.1.3 Non-concurrency Analysis using the PCFG and PSSA

Non-concurrent code is a subset of "race free" code. If we can statically analyze parallel programs for non-concurrent code, we can also eliminate these code segments from the instrumentation required for dynamic analysis. The non-concurrency analysis algorithm is shown in Algorithm 1 and is based on the method identified by Lin [71].

Algorithm 1 Non-Concurrency Analysis Algorithm [71]

```plaintext
for all nodes in PCFG do
    mark as non-visited
end for
partition a program into phases based on barriers
for each phase (barrier/parallel region) entry node N do
    call SearchforNon-ConcurrentPhase(N)
    mark all nodes as non-visited
end for

SearchforNon-ConcurrentPhase(N):
for each successor s of N do
    if s is not visited then
        if two nodes in a parallel region do not share any static phases then
            nodes = Non-Concurrent
            end if
        end if
        mark s as visited
        put s in same concurrent phase with N
        SearchforNon-ConcurrentPhase(s)
    end if
end for
end SearchforNon-ConcurrentPhase
```

The non-concurrency algorithm partitions a subroutine into phases based on barriers. Once the program is partitioned, each phase is checked for non-concurrency based on the
semantic constraints of the OpenMP constructs. If two nodes in a parallel region do not share any static phases, then the nodes are assumed to be non-concurrent. The semantics of \texttt{omp master}, \texttt{omp ordered} and \texttt{omp single}, prevent some statements from executing concurrently when multiple nodes are in the same phase.

### 8.1.4 Dead Code Analysis using the PCFG

Dead code is also a subset of “race free” code. Statically analyzing parallel programs for dead code can identify opportunities to eliminate these code segments from required instrumentation. The dead code analysis algorithm is shown in Algorithm 2.
Algorithm 2 Dead Code Analysis Algorithm

for all nodes in $PCFG$ do
    Mark all statements in a procedure FALSE and mark all OpenMP constructs TRUE.
end for

Let '$needed$' be the set of statements that return variables or output variables.

for each barrier/parallel region entry node $N$ do
    call SearchforDeadCode($N$)
end for

SearchforDeadCode($N$):

for each successor $s$ of $N$ do
    if '$needed$' is not empty then
        select and remove a statement $S$ from the set '$needed$' and mark it TRUE.
        for each variable $v$ in the statement $S$ do
            mark all statements in its ud-chain TRUE and add to the set '$needed$'
        end for
        if statement $S$ is an assignment of the form $v \leftarrow exp$ and du-chain of $v$ is an 'if' then
            mark it TRUE and add it to the set '$needed$'
        end if
    end if
end for
if all the statements in an OpenMP node are marked FALSE then
    mark the openMP constructs in the node as FALSE
end if
annotate all the statements and openMP constructs marked FALSE and empty nodes
end SearchforDeadCode

The dead code algorithm algorithm marks all instructions that compute needed values, where a value is needed if it is definitely returned or output by the procedure or it affects a storage location that may be accessible from outside the procedure. Then, the algorithm marks instructions that contribute to the computation of needed values. All unmarked instructions are considered dead code and can be annotated as such. In addition, variables that are only used to define new values for themselves are also considered dead. 

76
8.2 RaceFree Implementation

The implementation of RaceFree involves modifying the code of the OpenUH compiler to include the PCFG and PSSA representations and adding the nonconcurrency and dead code analysis algorithms.

Preliminary work has been performed to implement the PDFA framework into OpenUH. To create the PCFG, the WOPT is augmented while retaining the existing CFG structure and SSA code of OpenUH. An additional PREOPT phase was added to the WOPT, which we call PREOPT_OMP. This phase is invoked just prior to OpenMP Lowering and is analogous to the PREOPT_LNO/LNO compilation stage. In PREOPT_OMP, we perform the following steps:

1. **PCFG construction**: the CFG is traversed and the PCFG nodes/edges are constructed for identified parallel regions

2. **PCFG-PSSA construction**: an extended SSA representation is constructed on top of the CFG

3. **CFG-SSA Transformation**: transform back to an equivalent representation in the original CFG-SSA format

4. **SSA-based analysis**: leverage existing SSA-based analysis and implement non-concurrency and dead code analysis algorithms
Future work will continue the implementation of the PCFG and PSSA. Once completed, we can then add the non-concurrency and dead code analysis algorithms. The result of these algorithms will indicate the regions of code that can be eliminated from Thread Analyzer.

In the July 2008 build of Sun Studio Express, two new APIs \texttt{tha\_check\_datarace\_mem} and \texttt{tha\_check\_datarace\_thr} were added to allow OpenUH to interact with Thread Analyzer. The API \texttt{tha\_check\_datarace\_mem} instructs Thread Analyzer to ignore accesses to a specified block of memory when doing data race detection, whereas the API \texttt{tha\_check\_datarace\_thr} instructs Thread Analyzer to ignore memory accesses by one or more specified threads when doing data race detection \cite{101}. The use of these APIs during the implementation of RaceFree will enable the source code to be annotated as nonconcurrent and/or dead code that can be eliminated from the dynamic analysis of Thread Analyzer.

8.3 Measuring the Performance of RaceFree

A benchmark is needed to validate the strengths and limitations of RaceFree once it is fully implemented. The performance of RaceFree in combination with Thread Analyzer will be measured with SPECComp and NPB to identify any reduction in the overall duration to generate the final data race set. SPECComp and NPB will initially be used to compare the differences in duration with and without RaceFree. The use of RaceFree with Thread
Analyzer is expected to yield faster results on benchmark applications with large problem sizes.

A mature data race detection benchmark, with standardized evaluation criteria, currently does not exist within the High Performance Computing community. A future goal of this research will be to develop an OpenMP data race detection benchmark that can be used to objectively evaluate the performance of RaceFree and the complementary approach with Thread Analyzer. The benchmark will include seven different categories of faults that produce data race errors in OpenMP that were identified. Specifically, the benchmark will include a variety of errors in OpenMP syntax, synchronization (mutual exclusion, event, custom), data scoping and interprocedural/nested directives. As more static analysis capabilities are added to RaceFree, this benchmark will be useful to measure the data race detection effectiveness and efficiency of our solution.

Lu et. al, in their research with BugBench [74] [86], suggested bug detection tools should be evaluated by four criteria: functionality metrics, usability metrics, utility metrics and overhead metrics. Future performance measurement will indicate the completeness (finding all the bugs), scalability (handling millions of lines of code), soundness (confidence of error) and precision (few false positives) of the complementary data race detection approach and the usefulness of the RaceFree static analyses.
We choose to go to the moon in this decade and
do the other things, not because they are easy,
but because they are hard, because that goal will
serve to organize and measure the best of our
energies and skills, because that challenge is one
that we are willing to accept, one we are
unwilling to postpone, and one which we intend
to win, and the others, too.

John F. Kennedy

Chapter 9

Conclusion and Future Work

OpenMP provides an effective way to incrementally parallelize sequential programs. De-
spite the ease of using OpenMP, data race errors are common and can occur from the
inappropriate use of synchronization or inappropriate scoping of shared variables. Tools
that can efficiently locate and eradicate these errors will greatly enhance the productivity
of parallel software developers.

The contribution of this research was to present a complementary approach for OpenMP
data race detection using the OpenUH compiler and commercial dynamic analysis tools.
Experiments were performed to show the significant collection overhead of the dynamic
analysis tools. The use of static analysis in a complementary approach with the dynamic analysis tools can enhance the OpenMP parallel development debugging experience. Specifically, statically identifying non-concurrent and dead code, and annotating prior to dynamic analysis can reduce the collection time of dynamic analysis tools.

Proposed short-term goals for future work includes implementing RaceFree within the OpenUH compiler, creating an OpenMP data race benchmark suite, and measuring the improved performance of an end-to-end complementary approach to data race detection.

Longer term goals include static analysis research in enhancing RaceFree with the ability to perform escape analysis and belief analysis. Escape analysis can be used to determine if stack variables escape to the heap where another thread could access them. Variables that do not escape, do not need to be monitored, therefore reducing the instrumentation that is necessary. Finally, belief analysis can be used to create a method to prioritize the confidence of detected data race results, rating more likely data races over the less likely data races.
Bibliography


[47] D. P. Helmbold and C. E. McDowell. What is a Race in a Program and When Can We Detect It? Technical report, University of California at Santa Cruz, Santa Cruz, California, USA, 1993.


[92] J. Seward. Valgrind


89

