ABSTRACT

Title of Document: AUTOMATIC CRITICAL SECTION DISCOVERY USING MEMORY USAGE PATTERNS.

Lisa Marie Stechschulte, M.S., 2012

Directed By: Professor Donald Yeung
Department of Electrical and Computer Engineering

Parallel programming introduces new types of bugs that are notoriously difficult to find. As a result researchers have put a significant amount of effort into creating tools and techniques to discover parallel bugs. One of these bugs is the violation of the assumption of **atomicity**— the assumption that a region of code, called a **critical section**, executes without interruption from an outside operation.

In this thesis, we introduce a new heuristic to infer critical sections using the temporal and spatial locality of critical sections and provide empirical results showing that the heuristic can infer critical sections in shared memory programs. Real critical sections in benchmark programs are completely covered by inferred critical sections up to 75% to 80% of the time. A programmer can use the reported critical sections to inform his addition of locks into the program.

# AUTOMATIC CRITICAL SECTION DISCOVERY USING MEMORY USAGE PATTERNS

By

Lisa Marie Stechschulte

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master or Science
2012

# Acknowledgements

To Dr. Yeung, thank you for your creative ideas, challenging questions, and patient teaching.  I learned something new from you in every discussion.  Thank you.

To my committee, thank you for your flexibility and willingness to help me.

To my colleagues, thank you for supporting me through this writing process and covering for me while I was away working on my graduate studies.

To Adam, thank you for the support, encouragement, love, and home-cooked meals. I could not have done this without you… and I owe you dinner.

I dedicate my thesis to my parents.  You have always believed in me and told me I could do anything I set my mind to do.  Thank you.

# Table of Contents

# List of Tables

# List of Figures

## List of Code Segments

# Chapter 1: Introduction

To provide high performance, applications running on our home computers, web servers, data centers, and scientific computing clusters have embraced techniques to run many tasks at once. This move from serial to parallel processing allows a scientific application to run a complex computation faster, a web server to serve many clients at once, and our home computers to seamlessly offer high performance to multiple services at the same time.

However, writing an application that runs in parallel is difficult and prone to error. Programmers are trained to think of their code as running sequentially—one instruction follows the next and nothing is executed between them—but in a parallel program, two or more sections of the code that run simultaneously could interfere with each other. When that happens, a program could crash, hang eternally, or produce an incorrect answer.

Parallel programming bugs are notoriously difficult to find. It is not uncommon for a bug to manifest itself only with particular inputs and a specific interleaving of the simultaneously running sections of code. Researchers have put a significant amount of effort into creating tools and techniques to discover bugs in static and dynamic code and building systems that attempt to mask the effects of parallel bugs.

In this thesis, we present evidence for a new way to discover a type of parallel programming bug called an **atomicity violation**, which occurs when a programmer expects that a region of code will be executed without interference from other simultaneously executing code regions—that is it must execute **atomically**. A region of code that must execute atomically is called an **atomic section** or **critical section**. Existing research uses two primary methods to find atomicity violations. In one method, tools determine if code regions executing in parallel could be executed in some serial order and get the same result—in other words, determining if a parallel execution is **serializable**. If a parallel execution cannot be serialized, the tools assert that an atomicity violation may have occurred. The other method heuristically assumes that critical sections have temporal locality and that atomically executing large groups of consecutive memory accesses can prevent atomicity violations.

Our heuristic is quite different. We intuit that programmers want accesses to related variables to execute atomically. Thus, our tool forms **objects** composed of related variables and builds critical sections to protect accesses to these objects. In this study, we explain how we form objects and their related critical sections.

## *Parallel Computing Background*

To begin, we define **parallel computing** as performing two or more computations simultaneously. Each computation is performed by a **processing element**—a generic term to describe any unit that can perform a computation.

### Processing Element Organization

In his 1972 paper "Some Computer Organizations and their Effectiveness," Flynn classifies processing element organizations into four categories by how many instruction and data streams are used simultaneously by all processing elements. The simplest organization is Single Instruction-Single Data (**SISD**)—this is a sequential computer in which a processing element executes a single instruction a single data element at a time. When multiple processing elements execute the same instruction at the same time but operate on different data, they are using the Single Instruction-Multiple Data (**SIMD**) organization. A vector or array processor, where multiple processing units share a program counter but operate on independent data, is an example of SIMD organization. It is important to note that SIMD enforces processing elements to operate instructions in lockstep. The third organization, Multiple Instruction-Single Data (**MISD**), is not used frequently. In this organization, each processing element executes a different instruction on the same data. In the final organization, Multiple Instruction-Multiple Data (**MIMD**), each processing element executes a different set of instructions on a different set of data [9].

In later years, specific MIMD organizations were defined to include the Single Program-Multiple Data (**SPMD**) organization. In this organization, processing elements execute the same program, but do not operate in lockstep like the SIMD processing elements (each processing element has its own program counter). Data is divided among the programs [5].

## Memory Organization

In any Multiple Data organization, data can be distributed or shared among processing elements.

In the shared memory model, multiple processing elements have access to the same memory and can access the memory of every other processing element [6]. This is especially useful for communication among processing elements. In hardware, shared memory can be accomplished by having multiple processors on a single chip each with its own cache; all caches are then linked to a shared main memory. Cache coherence protocols keep each cache up to date with main memory and with the caches of the other processors as necessary. In software, the shared-memory model can be implemented using a multi-threading library like the *pthreads* library, for example. The library offers functions for the program to create a thread—a stream of instructions that can run independently of the main process but is also still sub-process of the main process. It also offers functions to synchronize the execution of the threads and provide threads with exclusive access to regions of memory through structures called *locks*.

In the distributed memory model, individual processing elements access only their own data and do not have access to the data of other processing elements. Processing elements can be sure that if they perform an operation on data, the data is current and has not been modified by any other processor. Communication between processing elements can be complicated in this model because it must take place in a separate network outside of memory. One communication solution is to use a message passing library like the Message Passing Interface (MPI) for communication between processing elements. To use MPI, the programmer writes a single program that runs on every machine in a cluster or every processing core of a single machine. MPI offers functions to send and receive messages and a barrier function, which stops processing on a core until a certain condition—for example, all cores have reported their results—is met. The programmer synchronizes the cores and communicates between them by using these send, receive, and barrier functions. All communication between cores in the message passing model is explicit, so the data on which one core is operating cannot be modified unexpectedly by another core [7].

## ***POSIX Threads*** **– The Parallel Implementation Studied**

For this study, we analyzed programs that used the POSIX Threads, or *pthreads*, C library to implement programs running the shared memory SPMD model. As the library's name indicates, threads are the fundamental tools used in this library to parallelize code. A thread is an independent stream of instructions that exists within a process. The thread uses some basic resources of its parent process, but the resources it uses are basic enough that the operating system can scheduled it independent of the parent process. While all threads have access to the program's global data, each thread maintains its own stack, registers (including an instruction pointer), and private data [18].

The *pthreads* library offers the function *pthread_create* for the main process or threads inside the main process to create new threads. When a thread is created, it is given a function to execute. This organization follows the SPMD model since every thread's code is part of a single program, but threads can execute different parts of the program simultaneously.

A thread can be terminated by the main process, another thread, or itself. The thread terminates itself when it returns normally from the function it was given at creation or when it calls the function *pthread_exit*. A thread can call *pthread_cancel* to terminated another thread. If the whole process is terminated or if *main* returns, all threads are terminated [18]. When a thread is terminated, its stack and private data are discarded.

Since all threads can read and modify the process's global data, the programmer must enforce that these accesses are appropriately synchronized. As an example, say a process has two threads that have finished their computations storing the results in *local_result* and now must add their result to the global variable *total*. Thread 1 finishes, reads *total*, locally performs the addition *local_total* = *local_result* + *total*, then stores *local_total* into *total*. Thread 2 will do the same tasks. If Thread 2 finishes just slightly after Thread 1 so that Thread 2 reads *total* after Thread 1 read *total* but before Thread 1 updated *total*, then the final result will not be *total* plus both variables *local_result*, but instead, will be *total* plus the *local_result* that updates *total* last. Access to *total* needs to be protected so that this cannot happen. This region of code is called a **critical section**, because it accesses a shared resource (*total*) and must execute **atomically**. Atomic execution means that all computations occur together without other threads interrupting. As Netzer and Miller put it, "Atomic execution means that the final state of variables read and written in the section depends only upon their initial state at the start of the section and upon the operations performed by the code (and not operations performed by another process)" [21].

The *pthreads* library provides the data structures and functions to synchronize access to global data and resources. The structure is called a **mutex**, which stands for "mutual exclusion." When a region of code must have mutually exclusive access to data or a resource, the programmer can guarantee this by insisting that before executing that region, the thread must **lock** the mutex using the function *pthread_mutex_lock*. Only one thread can hold a lock on a mutex at once. When mutually exclusive access to the data or resource is no longer needed, the mutex is

**unlocked** using *pthread_mutex_unlock*. It is then available for other threads to lock and access memory regions exclusively [18].

Threads can synchronize with each other using *pthreads* library's *barrier* function and condition variables and functions. The *barrier* function forces each thread to stop executing until all threads reach the barrier. A condition variable works with a mutex to signal events to other threads. As an example, Thread 1 recalculates a value *result* every time the global variable *total* is updated while other threads can update *total*. Thread 1 could continuously poll *total* waiting for a change and recalculating *result* when it notices *total* has changed; or when other threads update *total* from within a critical section (while a mutex is locked), they could notify Thread 1 via the condition variable that *total* has changed, awakening Thread 1 and causing Thread 1 to start its computation. The condition functions that we find in our analysis include *pthread_cond_wait*, which blocks a thread and releases the mutex until the condition variable changes, and *pthread_cond_broadcast*, which broadcasts a signal to all threads that the condition variable has changed [18].

## *Common Parallel Programming Bugs*

While performing multiple computations at once provides performance gains, it also introduces a new set of potential programming bugs. In the paper "Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics," the authors noted four types of bugs found in concurrent (i.e., parallel) programs [15]. Each of these bugs is discussed separately below.

The authors studied 105 randomly selected concurrency bug reports from the open-source software applications MySQL, Apache, Mozilla, and Open Office.

MySQL (a database application) and Apache (a web server) are both server applications; Mozilla (a web browser) and Open Office (a suite of word processing, spreadsheet, and other office applications) are both client applications. They examined everything included in the bug reports (source code, patches, programmer comments) to determine the type of bug and how it was fixed. From this, they developed an understanding of what types of bugs exist in the real world, under what conditions they appear, and how they are fixed [15].

## Deadlock

Deadlock occurs when two or more processing elements cannot proceed because they are waiting on each other to release access to a resource [15]. For example, Thread 1 must lock mutex A and then mutex B; meanwhile, Thread 2 locks mutex B and then mutex A. If simultaneously Thread 1 locks mutex A and Thread 2 locks mutex B, then neither will be able to proceed with their next lock and they will wait indefinitely.

It might seem that deadlock cannot occur when a program has only one gatekeeper guarding exclusive access to resources; however, in the "Learning from Mistakes" study, the authors found that 22% of deadlock bugs occurred because one thread tried to acquire a resource that it already had. They also found that 97% of the deadlock bugs were the result of "two threads circularly waiting for at most *two resources*" [15]. Despite the fact that increasing the number of exclusively accessed resources provides more opportunity for deadlock, very few of the deadlock bugs involved more than two resources.

## Data Race

A data race occurs when two or more processing elements access a shared variable without synchronizing [15]. The example that begins on page 5 where two threads add their results to a global variable *total* is an example of a data race. Here, the two threads race to update *total*, with the thread that updates *total* last overwriting the update made to *total* by the thread that updated *total* first.

Data races are not always errors. Sometimes programmers use data races as an efficient method of raising a flag to all processing elements that a condition has been met. For example, multiple processing elements are tasked with answering the question, "Is *x* in my data set? If so, where is *x*?" All processing elements share a global variable *found*, which is initially set to zero. Each processing elements checks *found* to see if it is non-zero indicating *x* has been found. If so, the processing element quits. If not, the processing element continues looking. If a processing element finds *x*, it updates *found* to the location of *x* without locking *found*. Because processing elements do not lock *found*, it is possible that two processing elements could simultaneously race to overwrite *found* with different locations; however, both locations are valid locations of *x*, so either location correctly answers the problem.

Because data races are not always bugs, the authors of "Learning from Mistakes" did not include them in the bug patterns they studied. Still many researchers have studied the area of data race detection and prevention, and we will discuss some of these studies in Chapter 2. It is important to note that a data-race free program may have other concurrency bugs.

## Atomicity Violation

As explained in the section above on *POSIX Threads*, atomic execution means that only the initial state of variables and data on a processing element and the instructions that processing element executes affect the final state of the variables and data [21]. The region of code that should execute atomically is called a **critical section**. If another processing element influences the final state of a critical section, then an atomicity violation has occurred. This occurs because the programmer incorrectly assumes that a section of code will execute atomically and does not protect the region from outside access.

As an example, consider a program in which the operations of Figure 1 on the global variable *pointer* occur on two different threads without any protection from outside accesses. In the code for Thread 1, the programmer incorrectly assumes that checking that *pointer* is initialized and dereferencing *pointer* will occur atomically. Instead, Thread 2 interferes with these two operations and resets *pointer* to NULL causing Thread 1 to crash when it tries to dereference *pointer*. The programmer could have fixed this bug by surrounding each block of code—each critical section—with a lock and unlock to a common mutex.

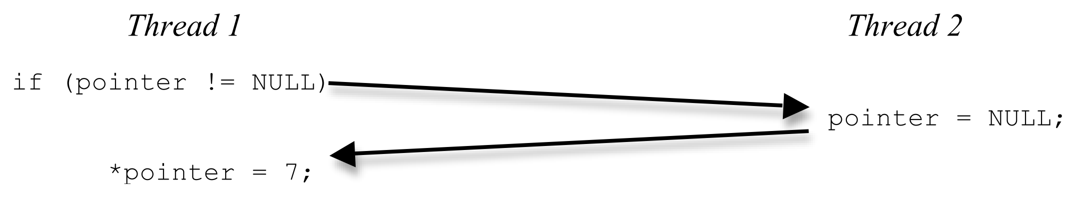


**Figure 1: Atomicity bug example.**

In the "Learning from Mistakes" study, 51 out of 74 non-deadlock concurrency bugs studied (nearly 69%) were atomicity violations [15].

## Order Violation

The authors of “Learning from Mistakes” found that order violations constituted 32% of the non-deadlock concurrency bugs examined. Unlike the three bug types discussed already, very little research has been done on detecting order violations [15].

An order violation occurs when the programmer assumes an order between blocks of code in different threads but does not enforce this order between the threads [15]. For example, the first thread initializes a variable and the second thread uses the variable. If the programmer does not force the initialization to occur before the second thread starts to use the variable, then an order violation occurs. In a server application where the main thread creates a new thread to handle each new client connection, this type of order violation could occur if the programmer starts the new thread before initializing a structure containing the client’s data.

A variety of fixes are available for this order violation. The programmer could insert a barrier function at the beginning of the second thread and after the initialization in the first thread; this requires that the second thread wait for the first to finish before continuing. The programmer could initialize the variable prior to creating the second thread, so that the second thread can proceed immediately once it is created. The second thread could poll the variable to see if it has been initialized and proceed as soon as the variable is initialized.

Note that only one of these solutions, the barrier solution, uses any sort of synchronization function. One finding of the “Learning from Mistakes” study was that 73% of non-deadlock concurrency bugs (of all types, not just order violations) were fixed without using locks [15]. While locks can fix this type of bug, they also can hurt performance making other fixes more efficient.

## *Motivation and Goals*

The "Learning from Mistakes" study showed that 69% of non-deadlock concurrency bugs examined were atomicity violations. Given that atomicity violations are so common, our goal was to determine if the memory access pattern of a dynamically executing program could suggest related variables that must be accessed atomically and where the critical sections for accessing these related variables were in the program's static code. If in fact the memory access pattern correctly discovered critical sections, this information could be used to build a tool that suggested potential critical sections to the programmer or a tool that automatically locked down critical sections. By automatically adding locks to critical sections, we could address the 27% of non-deadlock concurrency bugs in the "Learning from Mistakes" study that were fixed by adding or changing locks [15].

Of all non-deadlock concurrency bugs (atomicity violations, order violations, and other bugs) examined in the "Learning from Mistakes" study, 34% of them involved accesses to multiple (often related) variables (the remaining 66% involved only one variable); however, few concurrency bug detection tools look for bugs caused by accessing multiple variables [15]. Rather than using program variables to find the shared resources of critical sections, we aimed to address the multiple-variable nature of concurrency bugs by analyzing shared resources at the level of the memory layout. We formed **objects**, contiguous blocks of memory, by merging nearby memory addresses that are accessed close together in time; consequently, if access to multiple variables causes a concurrency bug and these variables are accessed nearby in space and time, they will form an object and can be detected as needing protection in a critical section.

## *Contributions*

Tools that detect and avoid atomicity violations are an important part of atomicity research. While all the tools address the problem slightly differently, they also use similar techniques. In Chapter 2: Related Work, we discuss several of these tools: AVIO [16], MUVI [14], Atom-Aid [17], and AtomTracker [20].

The major contribution of this study is the addition of a novel heuristic for determining critical sections. The heuristic is based on the idea that programmers assume that accesses to related variables happen atomically. We develop a method using the temporal and spatial locality of memory accesses to group related variables into memory objects and from those objects produce a set of static code critical sections inferred during a single run of a program.

This study gives empirical results showing that the novel heuristic can infer objects and critical sections in shared memory programs. A comparison with the real critical sections in analyzed programs shows real critical sections are covered by inferred critical sections up to 75% to 80% of the time. A programmer can use the reported critical sections to inform his addition of locks, or a new tool could be developed to automatically insert locks into the program for future runs.

Critical sections in this tool are inferred on the fly eliminating the need to collect massive memory traces required by AVIO and AtomTracker.

Programmers do not need to annotate code in any way in order for the tool to discover critical sections. In fact, the tool can run on a program that has no synchronization or locking implemented. Additionally, the code never needs to run correctly for the tool to infer critical sections. This is a prerequisite for AVIO and AtomTracker.

Like MUVI, our tool can detect critical sections that arise from using multiple variables. AVIO and Atom-Aid cannot do this.

The tool is implemented as a C++ Pintool (see Chapter 3: Inferring Critical Sections). This allows it to run on any program using X86 assembly language. Otherwise, the underlying hardware is irrelevant to the tool, unlike Atom-Aid, which must be run on top of an *implicit atomicity* system.

Perhaps the most important difference between this tool and AVIO, MUVI, Atom-Aid, and AtomTracker is that the only inter-processor information that our tool must know is whether a memory object is shared, in other words whether it is accessed by more than one processor. All the other tools require significant information about memory access interleavings. As a result, our tool can operate without significant inter-process communication.

# Chapter 2: Related Work

In this chapter, we review the current state of atomicity research. We begin by discussing transactional memory, an increasingly popular way of implementing critical sections without locks. Transactional memory grew out of the concept of database transactions and is used in parallel processing to avoid atomicity violations; we discuss several transactional memory studies below. Then we cover a series of studies that aim to automatically detect and correct atomicity violations.

## *Transactional Memory*

In the section *POSIX Threads* – The Parallel Implementation Studied, we introduced a lock-based system for implementing parallelism. However, another important shared-memory model for parallelism uses the concept of a **transaction**, which was developed in the study of databases. A transaction is a group of operations that optimistically execute atomically, meaning that the transaction will execute but before modifying the state of the shared memory or database will check to ensure that its own atomicity has not been violated by other transactions. If its atomicity has been violated, it aborts; otherwise, it commits its changes making them visible to the whole system [12]. In databases, the state of the system is allowed to be inconsistent while a transaction is executing, but must return to a consistent state once the transaction completes [8].

An example of a transaction is transferring money between bank accounts; here the constraint for the consistent state is that the amount of money stays constant. During the transaction, one bank account will be debited before the other is credited violating the constraint that the amount of money in the system is constant; however, at the end of the transaction, both bank accounts have been modified and the total amount of money is unchanged [8].

Transactions also must be **serializable**, meaning that even though transactions execute simultaneously, one can always get the same result by executing the transactions serially in some order—transaction operations cannot interleave [12]. As Eswaran et al. note, the serializability property is different from determinism—a set of transactions can be serialized without producing the same state in every execution of those transactions. In their paper "The Notions of Consistency and Predicate Locks in a Database System," they provide an excellent example of serializable transactions: consider an airplane reservation system where a transaction is assigning a seat for a reservation; the seats may be assigned differently depending on the serializable order of the transactions, but a consistent seating assignment will always be produced (no seat will be assigned more than once) [8].

In their 1993 paper "Transactional Memory: Architectural Support for Lock-Free Data Structures," Herlihy and Moss building on the database concept of transactions present the concept of *transactional memory*. Transactional memory relies on the concept of lock-free shared data structures, a data structure that does not require operations on it to be mutually exclusive. A lock-free data structure allows other processors to operate on it even if the first processor to operate on it gets interrupted during operation; in lock-based systems, this first processor would continue to hold the lock on the structure while it handles the interruption preventing all other processors from operating on it—this is called *convoying* [12]. In addition to preventing convoying, transactional memory has two other major advantages over lock-based models: transactional memory avoids the problem of *priority inversion* where a lower-priority process holds a lock needed by a higher-priority process preventing the higher-priority process from progressing. And, transactional memory cannot *deadlock* (see Deadlock above).

While others built software implementations of lock-free shared data structures, Herlihy and Moss created a new multiprocessor architecture to handle lock-free data structures. They added operations to the cache-coherence protocols to accommodate a set of new memory instructions that allow the programmer to specify that reads and writes to memory are included in a transaction and a set of instructions to change the transaction's state. The hardware to support transactions required two caches—a regular cache to handle non-transactional operations and a transactional cache to handle transactional operations. The transactional cache did not propagate writes to main memory unless a transaction successfully committed [12].

Herlihy and Moss compared their architecture with two software and two hardware methods for atomically updating memory. The two software methods and one hardware method were lock-based. The other hardware method used the LOAD_LINKED/STORE_COND operations from the MIPS II architecture, in which the LOAD_LINKED operation makes a local copy of a shared variable and only stores it back using STORE_COND if the shared variable has not changed since it was first read; essentially, this is a transaction with a single variable. The transactional memory architecture outperformed all four competitors.

In Herlihy and Moss's transactional memory architecture, the programmer is responsible for annotating transactions at the level of individual loads and stores. In the 2004 paper "Transactional Memory Coherence and Consistency," Hammond et al. eliminate the need for the programmer to annotate individual loads and stores by insisting that "transactions are always the basic unit of parallel work, communication, memory coherence, and memory reference consistency" allowing the programmer to insert only transaction boundaries [10]. They call their shared-memory model *Transactional Memory Coherence and Consistency* (TCC) and developed a hardware-based TCC that requires a central authority to regulate commits for the whole system and relies on broadcast communication of transaction commit packets. Like Herlihy and Moss's transactional memory hardware, TCC requires changes to the caches and cache coherence protocol to maintain transactions.

Programming for the TCC model requires first dividing the code into transactions, carefully ensuring not to break up critical sections. The hardware ensures that transactions are executed atomically, so the programmer does not need to worry about transaction independence. Optionally, the programmer may specify the order in which transactions should execute by giving transactions phase numbers. All transactions with the same phase number will be permitted to execute simultaneously, but TCC will not progress to the next largest phase number until all transactions at the current phase have committed.

In performance tests, TCC performed well, but required a high broadcast bandwidth for broadcasting commit packets. Because broadcast is central to TCC, TCC has limited scalability.

## *Atomicity Violation Detection and Prevention*

Transactional memory systems like lock-based systems are subject to atomicity violations. In the transactional memory systems described above, the programmer is responsible for correctly annotating transactions so that no critical section is split between two transactions. Likewise, in systems that use *POSIX* Threads, avoiding atomicity violations relies on the programmer's ability to recognize and protect critical sections. The following set of papers address detecting and preventing atomicity violations and identifying critical sections for the programmer.

Up until Lu et al.'s 2006 paper on "AVIO: Detecting Atomicity Violations via Access Interleaving Invariants," most concurrency bug detection research had focused on data race detection. As Lu et al. note, transactional memory models are not immune to atomicity violations, and the growing research in that field made addressing atomicity violations even more pertinent. Two years later their evaluation of the importance of detecting atomicity violations was reinforced by the "Learning from Mistakes" paper (which shared two co-authors with the AVIO paper) that showed atomicity violations constituted 67% of all examined non-deadlock concurrency bugs [15].

AVIO makes three contributions to atomicity violation detections. First, it uses "Access Interleaving (AI) Invariant based detection." Access Interleaving Invariant holds for an instruction if during all correct runs, there is no unserializable access between this instruction and the previous instruction that accessed the same shared variable. In a large number of correct runs, many possible correct interleavings will appear; if the Access Interleaving Invariant holds across all correct runs, then AVIO concludes that the programmer assumed these accesses would be atomic [16].

The authors' second contribution was building and analyzing both software and hardware versions of AVIO. They built the hardware version by extending the cache coherence protocol with the intention of using it to detecting atomicity violation during "production runs." The software version, which had more overhead than the hardware version, was intended for use as a debugging tool allowing a programmer to find a bug that has already occurred [16].

The final contribution was testing AVIO on six real atomicity violation bugs from server applications. They found that AVIO could detect a wider variety of bugs than previously tools. It did not report benign data races (see the section Data Race for a discussion of benign data races). It found only 3-5 static false positives compared with 80 or more false positives from other tools. It did not require the programmer use a specific parallel programming model or to annotate the program in any way. It did not require a large body of training data. The overhead created by AVIO for both the hardware version and the software version was smaller than that for other concurrency bug detection tools [16].

While AVIO only detects Access Interleaving Invariants for single variable accesses, MUVI suggests improvements on AVIO that would discover multi-variable accesses atomicity violations. MUVI uses static program analysis and data-mining techniques to determine correlated variables within large programs. In MUVI, correlated variables must be accessed with a common lock to avoid data races. Detecting atomicity violations is much harder. With the set of correlated variables provided by MUVI, AVIO can check the Access Interleaving Invariant between correlated variables. Checking for serializability is much more complicated with two or more variables and the authors of “MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs“ do not provide detail [14].

In 2008, Lucia et al. presented the first paper on surviving atomicity violations: “Atom-Aid: Detecting and Surviving Atomicity Violations.” Their paper begins with analyzing *implicit atomicity* systems—those in which the processors form arbitrary chunks of dynamic instructions and execute them atomically. This means that individual memory instructions within a chunk are never interleaved individually with instructions on other processors; instead, chunks are interleaved. Because chunks are updated atomically, an individual processor can reorder the instructions inside a chunk without influencing other processors. Examples of implicit atomicity systems include [2], [25], and [26]. Lucia et al. studied the effect of changing the chunk size on hiding atomicity violations. They found that while a larger the chunk size does correspond with a higher probability of hiding atomicity violations, the relationship is nearly logarithmic—while large probability gains are reported from increasing the size of chunks from 0 to 2000 instructions for atomicity violations with fewer than 750 instructions, larger chunks barely improve the probability of hiding the violation [17].

The authors sought to improve these probabilities by creating “smart” chunks rather than arbitrary chunks. Atom-Aid, which can be implemented in any implicit atomicity system, was the result. On the fly, Atom-Aid creates smart chunks by finding potential atomicity violations and inserting chunk boundaries at the beginning of the potential violations. It detects atomicity violations by looking for two accesses to the same variable *a* within a chunk on one thread with a recent access to *a* in a different thread from a committing chunk. If these accesses may be unserializable, then Atom-Aid watches for *a*, and when it appears again, Atom-Aid may insert a chunk boundary before the access to *a*. We use "may insert a chunk boundary" instead of "will insert a chunk boundary" because if Atom-Aid always broke a chunk at an instance *a*, it could actually expose the atomicity violation. Instead, Atom-Aid insists that a chunk will be broken at most once and the new chunk will be the default chunk size, and that if a new address is added to Atom-Aid's watch set, that it cannot break the current chunk, only later chunks [17].

Note that Atom-Aid requires knowledge of not only the current memory accesses from the local chunk, but also the memory accesses from the previous local chunk and the memory accesses from chunks committing elsewhere in the system. By bounding the stored memory accesses by chunks, Atom-Aid is able to reduce the amount of data it must manage and analyze. The cost of the boundary is that Atom-Aid cannot detect atomicity violations that are larger than two chunks; on the other hand, no implicit atomicity system can hide an atomicity violation that is larger than a single chunk [17].

Additionally, Atom-Aid is capable of starting a new chunk only after it has identified a variable as part of a potentially unserializable group of accesses [17]. This means that the first potentially unserializable group of accesses could form an atomicity violation that Atom-Aid cannot hide.

In testing, Atom-Aid performed well, hiding almost 100% of the atomicity violations in bug kernels and MySQL, Apache, and XMMS applications. However, for some applications, Atom-Aid created too many unnecessary chunks; for four of the nine bug kernels, the between 46% and 79% of the smart chunks created did not hide atomicity violations [17].

While Atom-Aid and AVIO sought to find and prevent atomicity violations, AtomTracker was built to discover generic atomic regions within programs without requiring any programmer annotation and then detect atomicity violations of these atomic regions. AtomTracker is split into two programs: AtomTracker-I that infers atomic regions and AtomTracker-D that tests for atomicity violations using these regions.

AtomTracker-I takes correct dynamic memory traces of a program and processes them one thread at a time greedily joining together consecutive memory accesses to form atomic regions. At the end, it outputs the entry and exit points of all atomic regions [20].

The algorithm works on a single thread at a time, looping through all threads for a single trace file first before proceeding to the next trace file. For Thread 1, AtomTracker-I begins by trying to merge the first two instructions into a single atomic region. For example, the first two accesses on Thread 1 are $I_0$, which reads *x*, and $I_1$, which writes y. If there are no conflicting accesses on any of the other threads, then $I_0$ and $I_1$ are merged into an atomic region. If $J_0$ from Thread 2 writes *x* and happens between $I_0$ and $I_1$, then $I_0$ and $I_1$ can be merged only if $I_1$ is brought up to $I_0$ and both happen before $J_0$; $I_0$ cannot happen after $J_0$ or else $I_0$ would not read the correct value of *x*. If instead $J_0$ from Thread 2 reads *y*, then $I_0$ can happen after $J_0$, but $I_1$ cannot happen before $J_0$ or else $J_0$ would read the wrong value of *y*. AtomTracker-I must perform this check of merging I0 and I1 into the same atomic region on every thread in the trace. Then it proceeds with the remaining instructions in Thread 1 [20].

Once AtomTracker-I has finished processing a single trace file, it then checks the atomic regions it created against the next trace file. If any are inconsistent, they will be split into smaller atomic regions. Once all trace files are processed, AtomTracker-I reports the resulting atomic regions [20].

Muzahid et al. note that atomic regions rarely cross loop iteration boundaries unless the full loop is included in the atomic region. Therefore, AtomTracker–I discovers loop boundaries and looks for conflicting accesses within the loop boundaries. If there are none, then it treats the entire loop as a unit incorporating either the whole loop or none of it into atomic regions. If there are conflicts within the loop boundaries, then AtomTracker-I ensures that if a loop iteration is included in any atomic region that the atomic region ends at the boundary of a loop iteration [20].

Finally, AtomTracker-I adds the atomic region boundaries it found to the static code of the program. Muzahid et al. point out that this can be particularly difficult when traces follow different control paths, so they supplement the exit points with their corresponding entry points to ensure that the correct exit is followed during a dynamic run using AtomTracker-D [20].

In addition to the inference tool, AtomTracker-D detects atomicity violations on the fly by running the program code modified by AtomTracker-I to contain the atomic regions. The tool works by determining if two concurrently running atomic regions can be serialized. If they cannot be serialized, then the program notes an atomicity violation [20].

Muzahid et al. implemented AtomTracker by building a C++ Pin tool (see Chapter 3: Inferring Critical Sections) and compared its performance with AVIO [16], MUVI [14], and PSet. The authors do not compare the atomic regions found by AtomTracker-I and the true atomic regions of a correct program; instead, they only report the atomicity violations discovered and the number of false positive atomicity violations discovered. AtomTracker successfully found eight atomicity violation bugs in the MySQL, Apache, and Mozilla, whereas AVIO found three, MUVI found four, and PSet found 2. The false positive rate ranged from 1.6 from the software implementation to 16.4 for a hardware implementation of AtomTracker-D.

# Chapter 3: Inferring Critical Sections

Determining memory objects that must be protected and code regions of a parallel benchmark program that should be treated as critical sections required the ability to dynamically analyze the benchmark's memory usage. While several tools can analyze a program's memory usage, we choose to use Pin because it provides an extensive API to dynamically instrument parallel Linux or Windows executables running on IA-32, IA-64, or Intel® 64 [22].

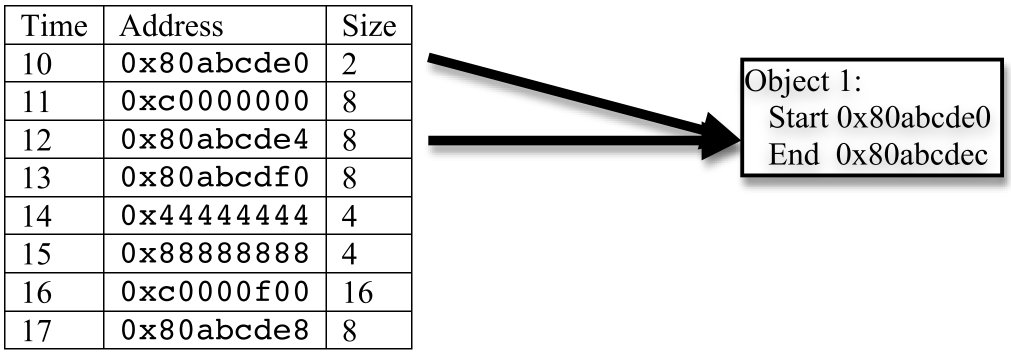
Instrumentation programs used with Pin, called Pintools, define insertion points and the actions that should be performed at those insertion points. Pin provides the API to insert instrumentation at different levels of a program including images, routines, basic blocks, and instructions. Actions at insertion points can include modifying the program's behavior by inserting C or C++ code or analysis that does not modify the program's behavior, such as saving every address accessed in memory for a memory trace.

For this study, we created a C++ Pintool that captured all non-stack and non-instruction memory operations (both reads and writes) to determine simultaneously if the address should be part of an object and, if it was part of an object, if the address should be included in a critical section. The benchmarks we analyzed used the *pthread* libraries for locking, so the Pintool also captured the *pthread_mutex_lock* and *pthread_mutex_unlock* functions to define the boundaries of the benchmark's real critical sections. The tool also discarded memory instructions that occurred during a call to *pthread_mutex_lock* to prevent analysis on locking-specific code; we wanted the tool to do the same for *pthread_mutex_unlock*, however, Pin could not identify the end of this function, so it was not possible to discard accurately all memory instructions that occurred during this call[1]. Once the program finished running, the Pintool computed statistics on how well the inferred critical sections covered the real critical sections and how many instructions were locked unnecessarily.

## ***Inferring Objects***

Every critical section has two components: a memory object and a region of code; the region of code must access the object atomically. Therefore, the Pintool must determine where the objects are before it can find any critical sections.

We define an object as a contiguous region of memory. If within a threshold amount of time (**time threshold**) a single thread of the benchmark accesses two memory addresses that fall within a threshold number of bytes (**address distance threshold**), the Pintool groups the two addresses into the same object. For example, consider the following memory accesses with a time threshold of 4 and an address distance threshold of 8 bytes:



| Time | Address | Size |
|---|---|---|
| 10 | `0x80abcde0` | 2 |
| 11 | `0xc0000000` | 8 |
| 12 | `0x80abcde4` | 8 |
| 13 | `0x80abcdf0` | 8 |
| 14 | `0x44444444` | 4 |
| 15 | `0x88888888` | 4 |
| 16 | `0xc0000f00` | 16 |
| 17 | `0x80abcde8` | 8 |

**Figure 2: Example memory accesses and the object created after the reference at time 12. Here the time threshold is 4 and the address distance threshold is 8 bytes.**

[1] Pin instruments calls to return to find the end of functions and does not guarantee success. Several other *pthread* functions are called by the benchmarks. These were not discarded in the Pintool's analysis.

Addresses 0x80abcde0 and 0x80abcde4 will be merged together since their time distance is 2 and their byte distance is 4. This will create a new object, *Object 1*, that starts at 0x80abcde0 and ends at 0x80abcdec (note that the end address is an exclusive upper bound on the object). Address 0x80abcdf0 will not be merged into this object since the byte distance to 0x80abcde0 is 16 and the byte distance to 0x80abcde4 is 12; note that this is still true after the *Object 1* is created.

The number of bytes accessed with an address is not considered in determining the distance between two addresses. In our example, address 0x80abcde4 with size 8 accessed at time 12 and address 0x80abcdf0 with size 8 accessed at time 13 will not be merged into an object even though the distance between the last byte accessed at time 12 (0x80abcdeb) and the first byte accessed at time 13 (0x80abcdf0) is only 5 bytes.

Once an object is formed, if an address inside of the object is accessed with a size that extends beyond the end of the object, the object's size will be increased to include all the bytes referenced by this access. Continuing our example from Figure 2, in Figure 3 we see the access at time 17 to address 0x80abcde8 is inside *Object 1*. The size of *Object 1* will be increased so the new end address is 0x80abcdf0.

| Time | Address | Size |
|---|---|---|
| 10 | 0x80abcde0 | 2 |
| 11 | 0xc0000000 | 8 |
| 12 | 0x80abcde4 | 8 |
| 13 | 0x80abcdf0 | 8 |
| 14 | 0x44444444 | 4 |
| 15 | 0x88888888 | 4 |
| 16 | 0xc0000f00 | 16 |
| 17 | 0x80abcde8 | 8 |

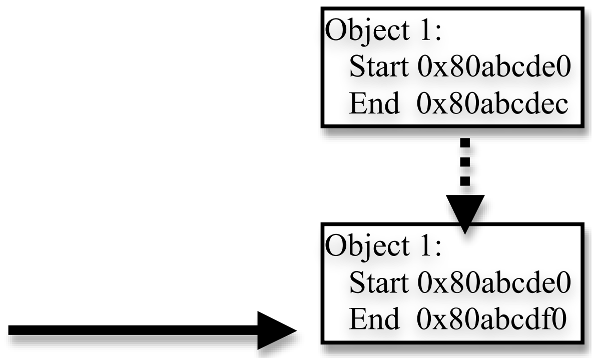


**Figure 3: Example memory accesses and modifying the object after time 17. Here the time threshold is 4 and the address distance threshold is 8 bytes.**

As we saw above when two addresses without objects are merged, they form a new object, and when one address that is in an object is merged with another that is not, the object containing the first address subsumes the second. If two addresses are already in separate objects are merged, their corresponding objects are also merged. For example, if the address accessed at time 10 was part of an object that started at 0x80abcdd0 and ended at 0x80abcde2 and the address accessed at time 12 was part of an object that started at 0x80abcde4 and ended at 0x80abce00, then the result of merging the addresses at time 10 and time 12 would be a single object that starts at 0x80abcdd0 and ends at 0x80abce00.

| Time | Address | Size |
|---|---|---|
| 10 | `0x80abcde0` | 2 |
| 11 | `0xc0000000` | 8 |
| 12 | `0x80abcde4` | 8 |
| 13 | `0x80abcdf0` | 8 |
| 14 | `0x44444444` | 4 |
| 15 | `0x88888888` | 4 |
| 16 | `0xc0000f00` | 16 |
| 17 | `0x80abcde8` | 8 |

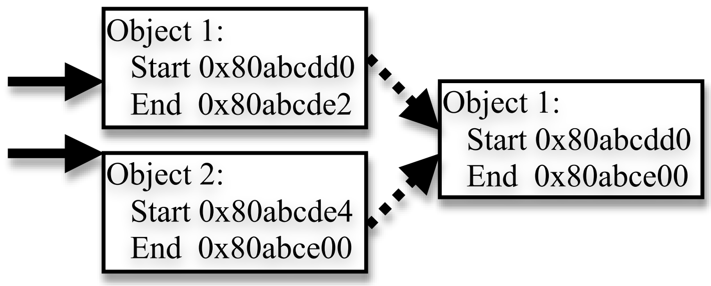


**Figure 4: Example memory accesses and merging after time 12 when both addresses are in objects. Here the time threshold is 4 and the address distance threshold is 8 bytes.**

Note that in this method of determining objects, a single address with no nearby accesses in space or time will never become an object. This "island" address may still need to be locked. Therefore, we developed a second method for creating objects termed the **island method**. In the island method, the first time a region of memory is accessed it becomes its own object; the starting address is the accessed address and the size of the object is the referenced size. After that, the merging process from the original method takes place.

Merging addresses into objects happens independently for each thread of the benchmark; however, once an object is created, it is maintained in a globally accessible bank of objects. This allows us to log whether an object is exclusive to the thread that created it or shared among multiple threads. An object that is never shared does not need to be locked.

Merging objects and maintaining the global bank is computationally intensive. As a result, we limited the memory operations used to create objects to those that accessed heap memory. Heap memory was determined by instrumenting all calls to *malloc* to capture the start address and the size of the allocated block. As discussed in Chapter 4: Results, nearly all of the benchmarks' real critical sections access heap memory, so narrowing the addresses we consider should not drastically impact our results.

## *Inferring Critical Sections*

At the same time the Pintool builds objects from the benchmark's memory accesses, it also infers critical sections. Although the Pintool only uses heap memory addresses to build objects, it examines all non-stack and non-instruction memory accesses while building critical sections. Critical sections are built by each thread independently and maintained independently until the benchmark finishes running.

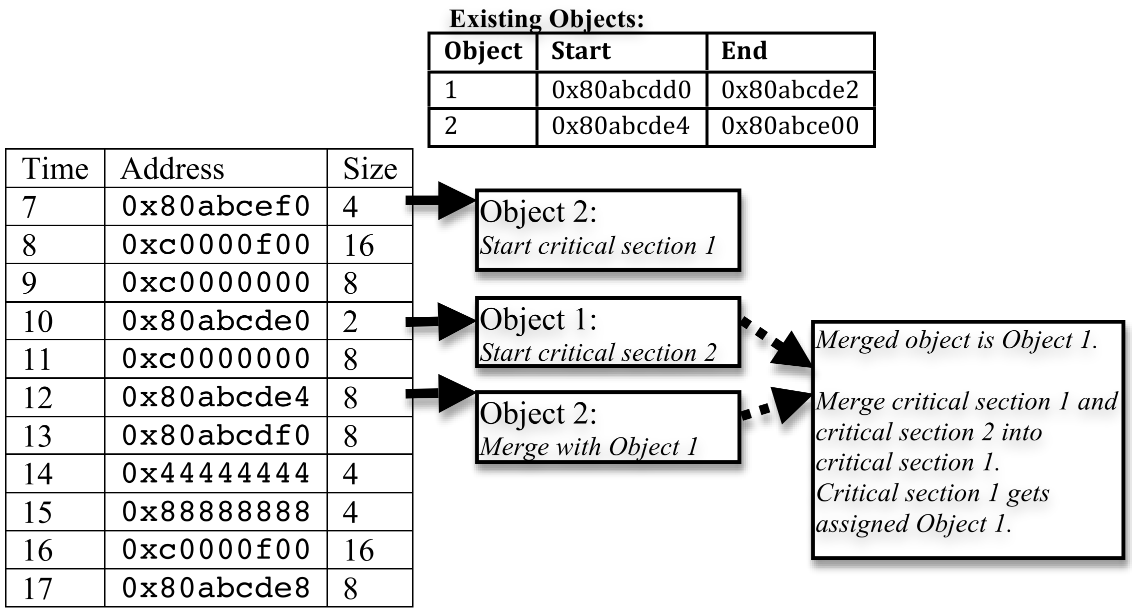
A critical section should begin when a thread first accesses an object and end when it last accesses that object before going on to do other tasks. Detecting the beginning of an inferred critical section is easy for the Pintool; if it accesses an object for which it is not currently creating a critical section, then that access begins a new inferred critical section for the object.

Detecting the end of a critical section is much harder. How does the Pintool know that a particular access to an object is the last to that object?[2] It cannot know; however, if it waits long enough and has seen no other accesses, it can guess that the critical section should have ended at the last reference. This is our strategy. The Pintool has a threshold for the number of memory accesses since it last saw the object accessed (**non-object accesses threshold**), and as soon as that many accesses have passed, it decides the critical section must have ended. The end of the critical section is the last reference to the object.

Inferred critical sections have two states in the Pintool. They are either active or finalized. An active critical section is one for which the Pintool has found a start, but has not yet found the end. Once the end is determined, the critical section is finalized.

A single thread can have multiple critical sections active simultaneously. This would not cause any confusion if the objects were statically determined prior to inferring critical sections, but objects can change while critical sections are active. If two objects are merged together while the Pintool has active critical sections for both of them, the two critical sections are merged also: the critical section that started first gets assigned to the new object; the critical section that started last is discarded. As an example, consider the sequence of memory references below. Two objects, *Object 1* that encompasses the access at time 10 and *Object 2* that encompasses the accesses at time 7 and time 12, have already been created:

[2] This question is similar to the concept of last-touch prediction for caches, and our solution is an algorithm similar to the Least-Recently Used algorithm for cache eviction.



**Existing Objects:**

| Object | Start | End |
|---|---|---|
| 1 | 0x80abcdd0 | 0x80abcde2 |
| 2 | 0x80abcde4 | 0x80abce00 |

| Time | Address | Size |
|---|---|---|
| 7 | 0x80abcef0 | 4 |
| 8 | 0xc0000f00 | 16 |
| 9 | 0xc0000000 | 8 |
| 10 | 0x80abcde0 | 2 |
| 11 | 0xc0000000 | 8 |
| 12 | 0x80abcde4 | 8 |
| 13 | 0x80abcdf0 | 8 |
| 14 | 0x44444444 | 4 |
| 15 | 0x88888888 | 4 |
| 16 | 0xc0000f00 | 16 |
| 17 | 0x80abcde8 | 8 |

Object 2:
*Start critical section 1*
Object 1:
*Start critical section 2*
Object 2:
*Merge with Object 1*
*Merged object is Object 1.*
*Merge critical section 1 and critical section 2 into critical section 1.*
*Critical section 1 gets assigned Object 1.*

**Figure 5: Merging active critical sections from when related objects are merged. Here the object creation time threshold is 4 and the distance threshold is 8 bytes. The critical section non-object accesses threshold is 50.**

Object 1 and Object 2 are merged together into Object 1. Since both have active critical sections, the two critical sections are also merged and assigned to Object 1; Critical Section 2 is then discarded. Due to the complexity of bookkeeping, finalized critical sections are not modified even if their corresponding objects get merged with new addresses or other objects.

Multiple threads can create critical sections for the same object at the same time. We allow this to ensure that the Pintool would catch all critical sections if a program had atomicity bugs.

When a thread finishes, the Pintool displays all critical sections that the thread found dynamically and whether the object for which the critical section was created was shared or exclusive to that thread. If the thread repeated execution of a particular region of code where it found a critical section once, it is possible that it found the critical section multiple times, and that critical section will get reported multiple times. Once all threads are finished, a complete list of critical sections found dynamically in all threads is created.

From this list, the Pintool weeds out critical sections that were found multiple times to determine the static critical sections. Static critical sections are the set of critical sections with unique start and end instruction pairs. If any dynamic instance of a critical section was built from a shared object, the static critical section is considered to reference shared objects also even if some dynamic instances of the critical section were referencing objects accessed exclusively by a single thread.

# Chapter 4: Results

While the method of inferring objects and critical sections may seem simple, applying it to real programs can lead to complex results. In this chapter, we discuss the benchmarks we tested, how we evaluated the tool's performance, the overall results for real critical section coverage and inferred critical section accuracy, and finally some specific examples of real critical sections that the tool did or did not find.

## *Benchmarks*

To test how well we could infer critical sections using a program's memory usage pattern, we ran the Pintool described above on six benchmarks from the SPLASH-2 benchmark library: Barnes, FMM, Ocean Contiguous Partition, Ocean Non-Contiguous Partition, Water-nSquared, and Water-Spatial. The SPLASH-2 benchmarks are all parallel applications designed for shared memory systems. The six implementations we tested all perform scientific computations, so these results may not transfer to non-scientific programs.

The SPLASH-2 benchmarks are written in C and use macros to define parallel constructs (locks, unlocks, barriers, etc.). For our study, we used the file c.m4.null.POSIX provided on the Modified SPLASH-2 Home Page to replace macros with the original POSIX Thread standard functions [18]. LOCK macros are replaced with a call to *pthread_mutex_lock* and UNLOCK macros are replaced with a call to *pthread_mutex_unlock*. The original POSIX Thread standard did not implement any barrier function, so instead barriers in c.m4.null.POSIX start with *pthread_mutex_lock* and check condition variables to determine if all threads have reached the barrier; finally, they end with *pthread_mutex_unlock* [18].

Of the six benchmark applications, there are three categories of computation. Barnes and FMM (Fast Multipole Method)—both categorized as *N*-body problems—do similar calculations to evaluate the interaction of bodies like particles or galaxies over time. For both, data are arranged in a tree structure, but Barnes traverses the tree once per body, and FMM traverses the tree once per time step [24]. Ocean Contiguous and Ocean Non-Contiguous both simulate ocean movements (eddies, currents) by partitioning data into grids. In Ocean Contiguous the grid is represented with three-dimensional arrays so that data are partitioned contiguously; in Ocean Non-Contiguous the grid is represented by two-dimensional arrays, which do not allow representing data contiguously [1]. Like Barnes and FMM, Water-nSquared and Water-Spatial are *N*-body problems. They both simulate the forces on a system of water molecules over time, but Water-Spatial uses an $O(n)$ algorithm while Water-nSquared uses an $O(n^2)$ algorithm [24].

The benchmarks have been thoroughly tested and in use for many years, so we assumed that they were bug free; thus the only critical sections in the benchmarks are those explicitly marked by *pthread_mutex_lock* and *pthread_mutex_unlock* functions.

Because of the computational complexity of the Pintool, in order to perform the analysis we had to use the smallest possible input data size. Reducing the data size to train an algorithm is common. The end results can still be used on a larger data set. The Appendix contains the input data for each benchmark. One input parameter for each benchmark is the number of threads the benchmark should use; this was set to four for all trials.

## ***How suitable is the tool for these benchmarks?***

As mentioned in Chapter 3, the computations needed to infer objects and critical sections were too complex to run on all memory references. Therefore, we reduced the number of memory references used for both object and critical section inferences by ignoring all stack-relative and IP-relative references. For the object inference, we had to further reduce the set of memory references to only heap references. If a program's real critical sections do not lock heap memory accesses, then the tool will never catch them.

To determine how suitable the benchmarks were to analysis under these restrictions, we created a small memory profiling Pintool. It caught all calls to *malloc* to determine the boundaries of heap memory. The boundaries of real critical sections were defined by calls to *pthread_mutex_lock* and *pthread_mutex_unlock*; the tool also logged the first and last memory accesses inside the lock to determine if the calls locked user code or library code. Finally, the Pintool calculated the number of heap and non-heap references in critical sections excluding IP-relative and stack-relative references. All results were reported for dynamic critical sections.

As Figure 6 shows, at least 80% of real critical sections in all benchmarks touch some element of heap memory. In fact, between 25% and 65% of real critical sections reference only heap memory.

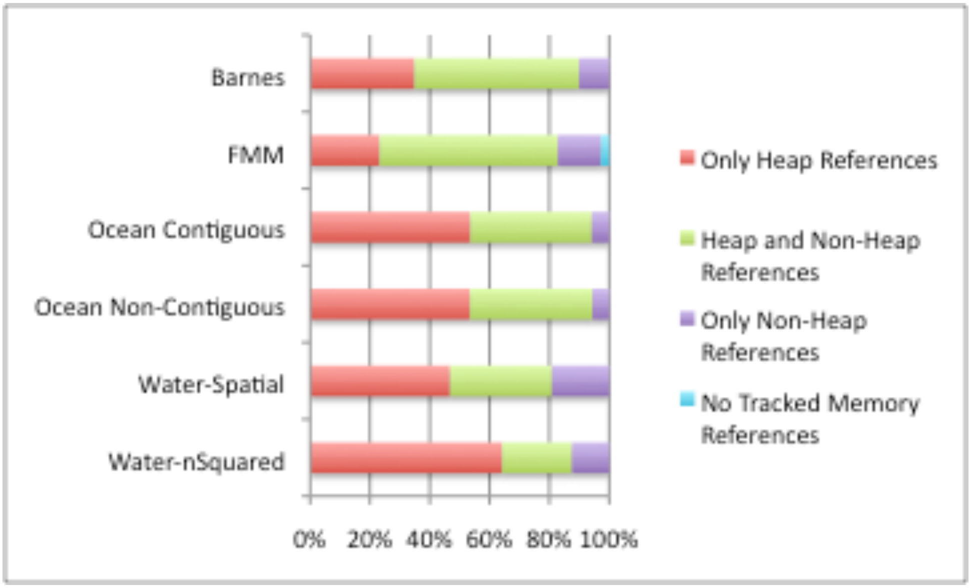


**Figure 6: Percentage of real critical sections that reference only heap memory, only non-heap memory, some of both, and no tracked memory references.**

Of the critical sections that do not reference any heap memory, we were able to use the first and last memory accesses inside the lock to determine if user or library code was being locked. Table 1 shows that every benchmark had 17 dynamic critical sections in library code that referenced non-heap memory exclusively. For every benchmark, 16 of these were in the *ld-linux-x86-64.so.2* library's procedure *_dl_fini* and the last one was in a *libc.so.6*'s procedure *_dl_addr*. These library functions are used to initialize and end a program or thread. For four of the six benchmarks (Water-nSquared, Ocean Non-Contiguous, Ocean Contiguous, and Barnes), these library critical sections were the only ones that referenced only non-heap memory.

**Table 1: Number of dynamic real critical sections that only referenced non-heap memory by code region.**

| | Barnes | FMM | Ocean Contiguous | Ocean Non-Contiguous | Water Spatial | Water-nSquared |
|---|---|---|---|---|---|---|
| **Library code** | 17 | 17 | 17 | 17 | 17 | 17 |
| **User code** | 0 | 9 | 0 | 0 | 1 | 0 |

Of the remaining two benchmarks, Water-Spatial had one large critical section from user code that referenced non-heap memory exclusively. It had 2919 references to non-heap memory and 0 to heap memory and is located in mdmain.C lines 172 to 177. When it is translated to use *pthread_mutex_lock* and *pthread_mutex_unlock* this segment of code is as follows:

```
/* if it is time to print output as well ... */
if ((i % NPRINT) == 0 && ProcID == 0) {
      {pthread_mutex_lock(&(gl->IOLock));};
      fprintf(six,"     %5ld %14.5lf %12.5lf %12.5lf %12.5lf \n"
            ,i,TEN,POTA,POTR,POTRF);
      fprintf(six," %16.3lf %16.5lf %16.5lf\n",XTT,AVGT,XVIR);
      fflush(six);
      {pthread_mutex_unlock(&(gl->IOLock));};
}
```

**Code Segment 1: Water-Spatial *mdmain.c* printing critical section.**

This critical section grabs the input and output lock to prevent any other thread from printing results between its calls to *fprintf* and *fflush*. The value *six* is a globally declared *FILE* pointer set to *stdout* at the beginning of the program. The remaining variables that are printed during this critical section (TEN, POTA, POTR, POTRF, etc.) are stack variables. Since none of the variables referenced here are dynamically allocated, our Pintool cannot capture this critical section.

FMM has nine dynamic user code critical sections that do not reference heap memory; of these, there are three static critical sections. One static critical section is in a function called *LockedPrint*, which behaves like the critical section in Water-Spatial above; this function locks the input and output lock, flushes *stdout*, prints the input arguments to *stdout*, and flushes *stdout* again. It does not reference heap memory, because the input arguments to the function are passed by value and not by reference.

FMM has two static critical sections—one in a function called *CreateParticleList* and the other in a function called *CreateBoxes*—that protect a single call to *malloc*. Because *malloc* is not a thread-safe function (functions that are not thread-safe are not guaranteed to execute correctly if control is switched from them while they are executing), it must be protected by a lock in order to ensure it executes correctly. As an example, the code from *CreateBoxes* is below in Code Segment 2. Although *malloc* is called here, the Pintool does not register a reference to heap memory.

```
{pthread_mutex_lock(&(G_Memory->mal_lock));};
Local[my_id].B_Heap = (box *) malloc(num_boxes * sizeof(box));;

/*
. . .
*/

{pthread_mutex_unlock(&(G_Memory->mal_lock));};
```

**Code Segment 2: FMM *CreateParticleList* malloc critical section.**

Finally, FMM is the only benchmark with dynamic critical sections that do not have any of the memory references we track. There are five dynamic instances of this one static critical section which is found in the function *InsertBoxInGrid*. The code for this critical section is below:

```
{pthread_mutex_lock(&(G_Memory->single_lock));};
if (Grid == NULL) {
      Grid = b;
      success = TRUE;
}
else
      success = FALSE;
{pthread_mutex_unlock(&(G_Memory->single_lock));};
```

**Code Segment 3: FMM *InsertBoxInGrid* critical section.**

*Grid* is a global pointer initialized to NULL, *b* is a pointer variable passed into the function, and *success* is a local variable. This critical section protects testing the value of *Grid* and if it has not been set, setting *Grid*. It accesses only the pointers and not the memory they reference. Therefore, no matter which path the program takes through the *if*-statement the result will be the same: no heap memory is accessed. In fact, this segment of code does not reference any memory that our tool tracks. The assembly code corresponding to the critical section is below:

```
403fbb:     cmpq   $0x0,2145213(%rip)        # 60fb80 <Grid>
403fc3:     je     404065 <InsertBoxInGrid+0x1e5>
403fc9:     mov    2197264(%rip),%rdi        # 61c6e0 <G_Memory>
403fd0:     add    $0x50,%rdi
403fd4:     callq  400e88 <pthread_mutex_unlock@plt>

. . .

404065:     mov    $0x1,%r13b
404068:     mov    %rbp,2145041(%rip)        # 60fb80 <Grid>
40406f:     jmpq   403fc9 <InsertBoxInGrid+0x149>
```

**Code Segment 4: Assembly code for FMM *InsertBoxInGrid* critical section.**

Here, *Grid* is referenced relative to the instruction pointer; in lines 403fbb and 404068, Grid is `2145041(%rip)`. In line 404068, *b* is referenced relative to the stack pointer (`%rbp`). The register `%r13b` stores the variable *success*. Our tool captures none of these references, so this critical section appears to have no memory references.

Handling stack, IP-relative, and non-heap memory accesses is important. However, based on these results, we determined that forming objects by analyzing only heap memory and discarding stack-relative and IP-relative references could provide a good initial demonstration of our heuristic. Future work could address expanding the analysis to other memory regions by increasing the efficiency of our Pintool.

## ***Measuring Success***

The Pintool takes several measurements to determine how well the inferred critical sections cover the real critical sections. Coverage is evaluated at run time to develop statistics on the dynamic coverage of real critical sections and at the end of the run to determine the static coverage.

The following categories of real critical sections coverage are tracked for both dynamic and static coverage:

- **Fully Covered**: every instruction in the real critical sections was covered by one or more inferred critical sections where none of the inferred critical sections ended during the real critical section. Because no inferred critical sections ended during the real critical section, at least one inferred critical section covered the entire real critical section.
- **Fully Covered with Overlap**: real critical sections that were fully covered by one or more inferred critical sections. Although these real critical sections are

fully covered, they could be covered by overlapping inferred critical sections, none of which completely covers the real critical section.

- **Partially Covered**: real critical sections that are partially covered by inferred critical sections. If one or more instructions of the real critical section are left uncovered, then the critical section is partially covered.
- **Uncovered**: real critical sections that are completely uncovered.

It is important to note that due to engineering constraints we had to calculate these statistics using all inferred critical sections instead of only inferred critical sections that locked shared objects. If the tool worked perfectly, all real critical sections would be *fully covered*. However, if the tool created a single inferred critical section that covered the entire program, the result would be that all real critical sections were *fully covered*, but the inferred critical section would be meaningless.

To judge how well the tool has done, we also need to understand how often inferred critical sections are created where they are not needed. The Pintool grouped inferred critical sections into three categories both dynamically and statically:

- **All Instructions Cover Real**: all instructions in the inferred critical section are also in at least one real critical section.
- **Some Instructions Cover Real**: some instructions in the inferred critical section are also in at least one real critical section.
- **No Instructions Cover Real**: no instructions in the inferred critical section are also in a real critical section.

Creating a critical section where one is not needed negatively impacts performance. To measure how much performance is affected by the inferred critical sections, the Pintool determined dynamically how many instructions in inferred critical sections are also in real critical sections and how many are not.

We ran the island and original methods of finding inferred critical sections on each of the benchmarks using a variety of thresholds. The results for static and dynamic real critical section coverage are in Figure 7 through Figure 30. The results for static and dynamic inferred critical section accuracy are in Figure 31 through Figure 54. In each graph, the thresholds for object and critical section creation fall along the y-axis; they are listed in the following order from left to right: time threshold for inferring objects, address distance threshold for inferring objects, non-object accesses threshold for inferring critical sections. Table 2 shows the tested threshold values; all 112 combinations of the three threshold values were tested.

**Table 2: Tested threshold values for all three thresholds.**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Time Threshold** | 2 | 4 | 8 | 16 | | | |
| **Address Distance Threshold** | 8 | 16 | 32 | 64 | | | |
| **Non-object accesses threshold** | 25 | 50 | 100 | 150 | 200 | 250 | 300 |

## *Real Critical Section Coverage*

Overall, the graphs show that in both the original and island method the tool covered real critical sections fairly well with the tested thresholds. Around 75% to 80% of static real critical sections were fully covered, fully covered with overlap or partially covered with the best set of thresholds for all benchmarks with the majority being fully covered. For dynamic real critical section coverage, this number jumps to around 90%.

Statically, the island method outperformed the original method for three benchmarks mainly because it fully, fully with overlap, or partially covered real critical sections regions that were uncovered using the original method. This trend is most noticeable in the figures for static critical section coverage for Barnes, Water-Spatial, and Water-nSquared (see Figure 12, Figure 11, Figure 18, Figure 17, Figure 16, and Figure 15). Dynamically, however, the island and original methods performed about the same. The almost unnoticeable differences in dynamic coverage suggest that the critical sections left uncovered by the original method are not executed much dynamically.

A prominent cyclic pattern appears in the graphs across all benchmarks in both static and dynamic coverage for original and island methods. When the address distance threshold is set to 8, the number of fully covered real critical sections is low and there are many real critical sections that are fully covered with overlap. When the address distance threshold is set to 16, 32, or 64, the number of real critical sections that are fully covered with overlap is reduced to almost none while the number that are fully covered increases to make up the difference.

The non-object accesses threshold did not affect the performance of the tool significantly. As the non-object accesses threshold gets larger, the number of fully covered with overlap and the number of partially covered real critical sections are reduced to become fully covered real critical sections. This trend is easiest to observe when the address distance threshold is 8 since that is when the tool reports the most fully covered with overlap and partially covered real critical sections.

The time threshold used to make objects has almost no affect on the tool's performance. The time threshold was only varied between 2 and 16 memory accesses, so the apparent insignificance of this parameter may be the result of an insignificant range of time thresholds tested.

Each particular set of thresholds performed consistently across all benchmarks. This suggests that the best performing thresholds found here could be used across similar types of scientific applications.

All the benchmarks show a constant number of static and dynamic critical sections across all threshold triplets except Barnes, where the number of dynamic critical sections changes for different sets of thresholds (see Figure 23 and Figure 24). Since the Pintool does not interfere with the execution of the benchmark, we do not expect that the benchmark would execute differently for different thresholds. In fact, the variation in the number of dynamic critical sections seen in Barnes is unrelated to the thresholds. The number varies every time Barnes is run. There are two critical sections in the function *loadtree* that get executed a non-deterministic number of times in every run. This function descends the tree data structure to insert new nodes and locks a parent node whenever a modification must be made to its children. Because each thread is executing this process simultaneously, specific thread interleavings can influence how often a modification to a parent node happens and thus how often a critical section is executed.

## Static Real Critical Section Coverage

🟦 Fully Covered 🟩 Partially Covered 🟥 Fully Covered with Overlap 🟪 Uncovered

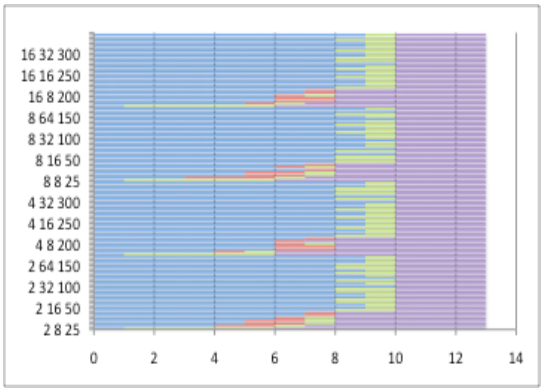


Figure 12: Static real critical section coverage for Barnes.

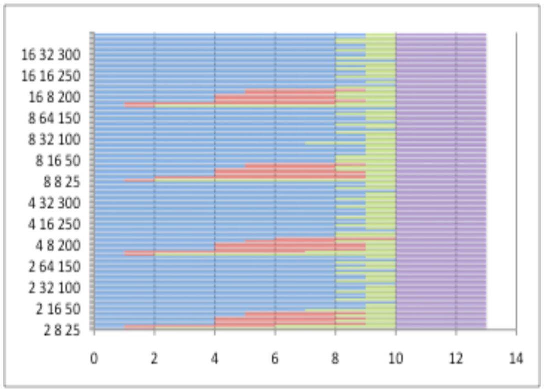


Figure 11: Static critical section coverage for Barnes using Islands.

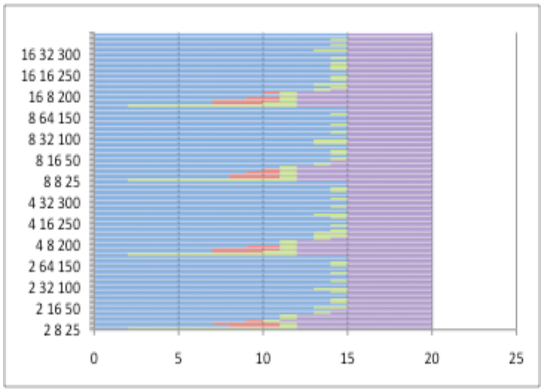


Figure 10: Static critical section coverage for FMM.

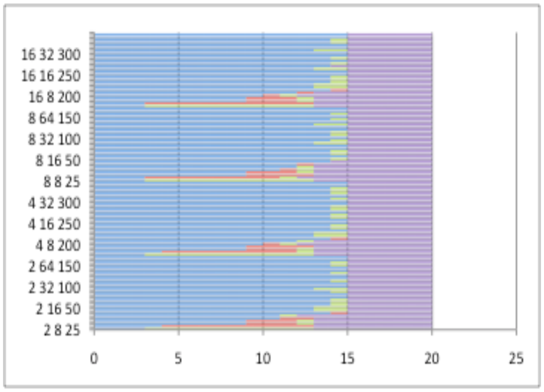


Figure 9: Static critical section coverage for FMM using Islands.

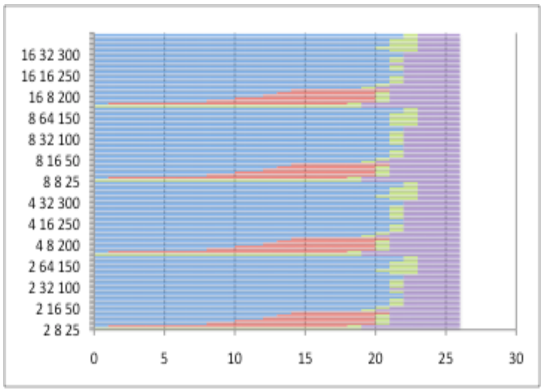


Figure 8: Static critical section coverage for Ocean Contiguous.

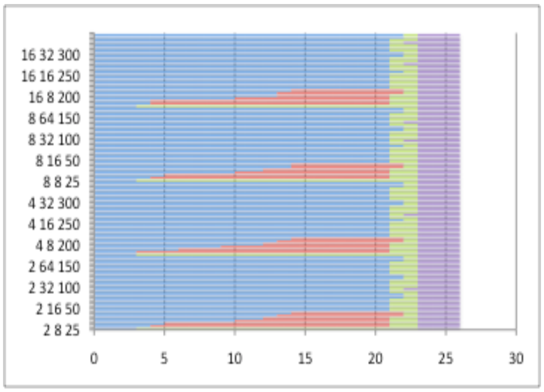


Figure 7: Static critical section coverage for Ocean Contiguous using Islands.

🟦 Fully Covered 🟩 Partially Covered 🟥 Fully Covered with Overlap 🟪 Uncovered

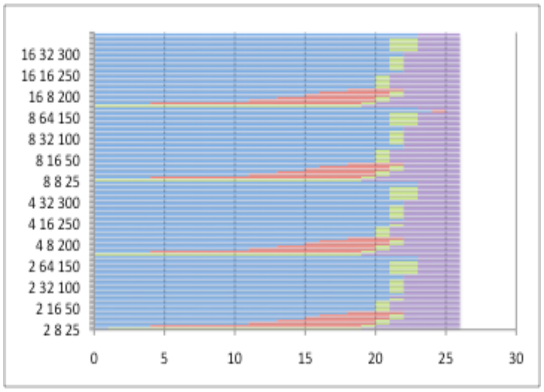


**Figure 13: Static critical section coverage Ocean Non-contiguous.**

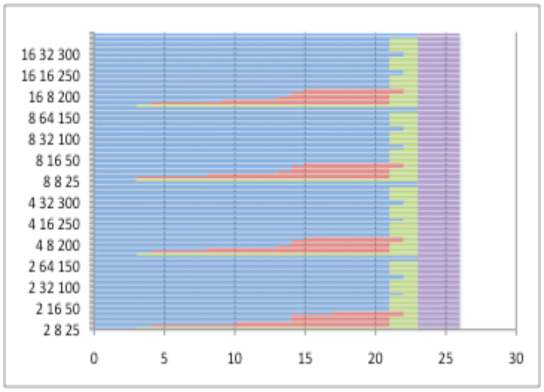


**Figure 14: Static critical section coverage Ocean Non-contiguous using Islands.**

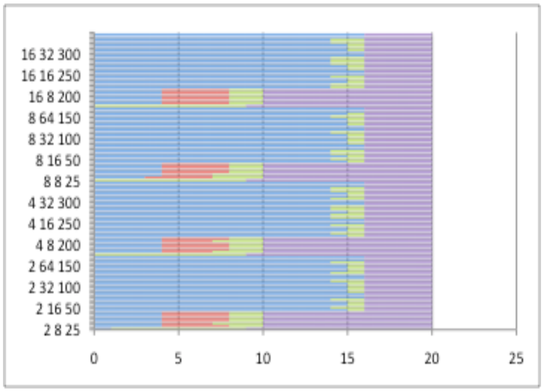


**Figure 18: Static critical section coverage for Water-Spatial.**

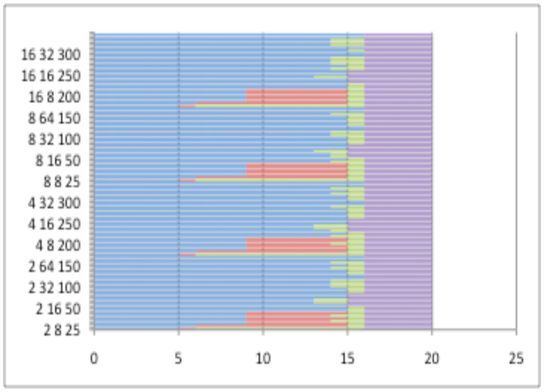


**Figure 17: Static critical section coverage for Water-Spatial using Islands.**

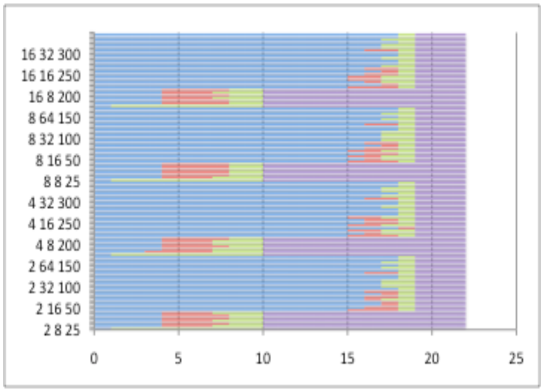


**Figure 16: Static critical section coverage for Water-nSquared.**

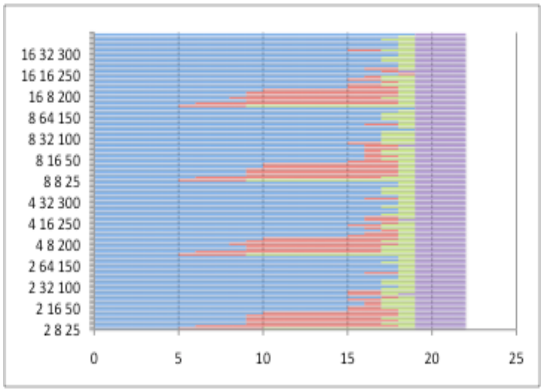


**Figure 15: Static critical section coverage for Water-nSquared using Islands.**

## Dynamic Real Critical Section Coverage

🟦 Fully Covered 🟩 Partially Covered 🟥 Fully Covered with Overlap 🟪 Uncovered

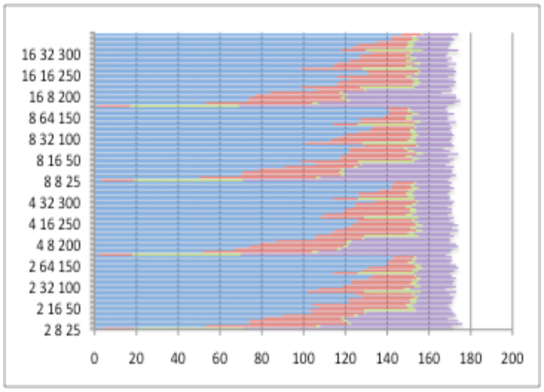


Figure 24: Dynamic critical section coverage for Barnes.

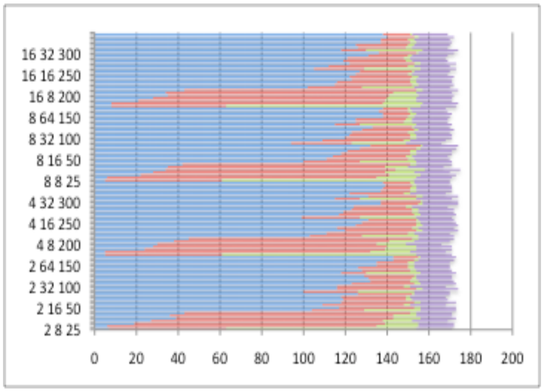


Figure 23: Dynamic critical section coverage for Barnes using Islands.

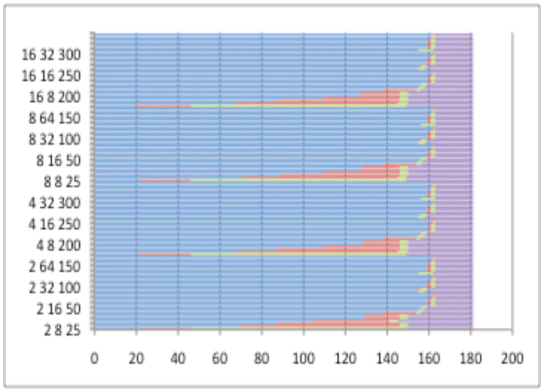


Figure 22: Dynamic critical section coverage for FMM.

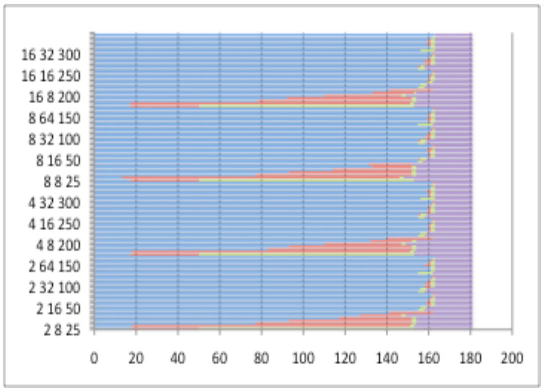


Figure 21: Dynamic critical section coverage for FMM using Islands.

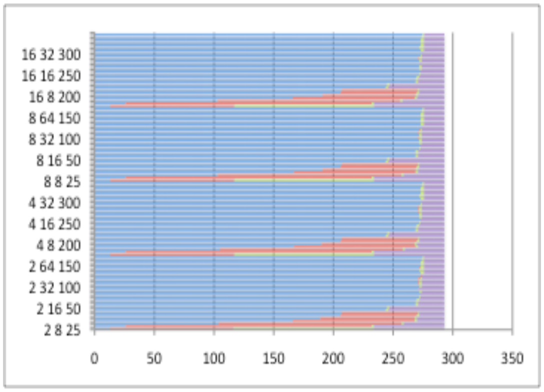


Figure 20: Dynamic critical section coverage for Ocean Contiguous.

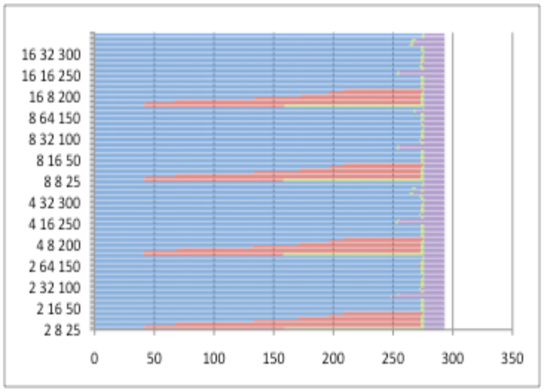


Figure 19: Dynamic critical section coverage for Ocean Contiguous using Islands.

🟦 Fully Covered 🟩 Partially Covered 🟥 Fully Covered with Overlap 🟪 Uncovered

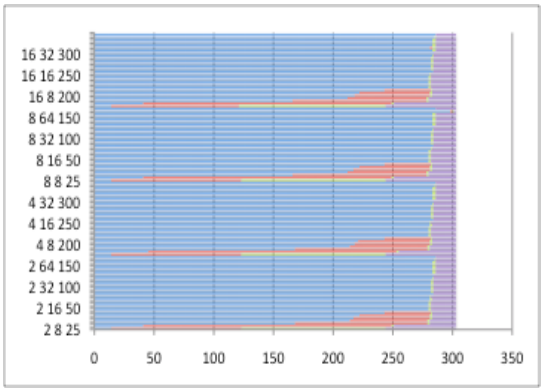


Figure 30: Dynamic critical section coverage for Ocean Non-contiguous.

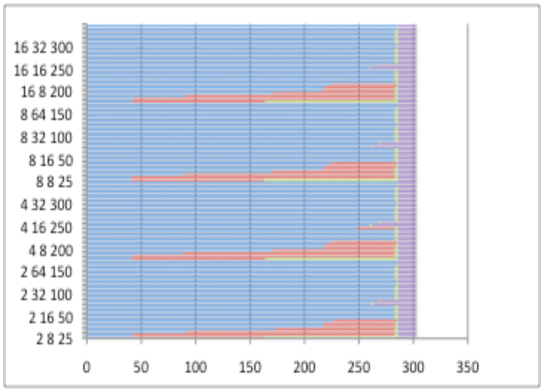


Figure 29: Dynamic critical section coverage for Ocean Non-contiguous using Islands.

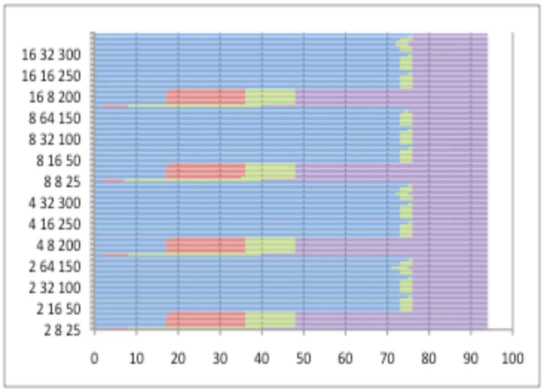


Figure 28: Dynamic critical section coverage for Water Spatial.

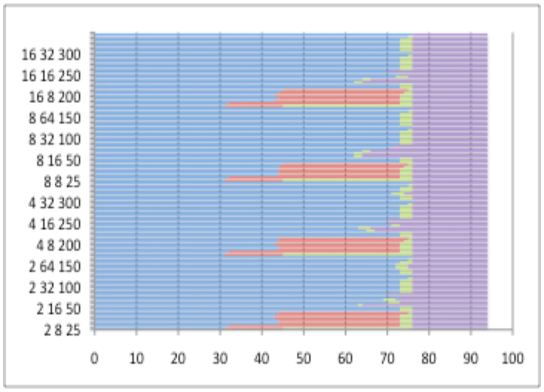


Figure 27: Dynamic critical section coverage for Water Spatial using Islands.

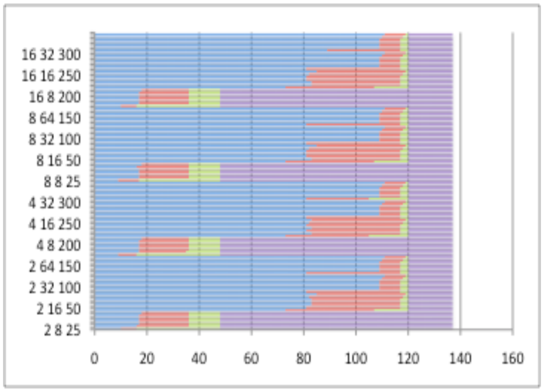


Figure 26: Dynamic critical section coverage for Water-nSquared.

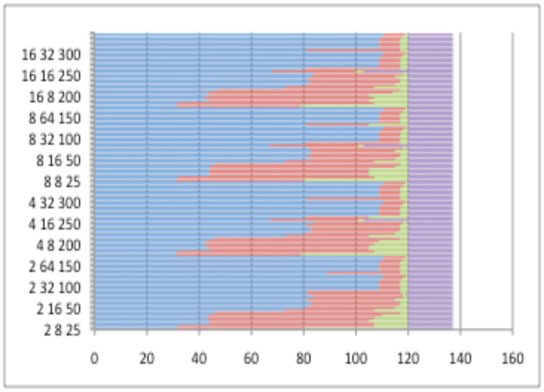


Figure 25: Dynamic critical section coverage for Water-nSquared using Islands.

## *Inferred Critical Section Accuracy*

While inferred critical sections covered 75% to 80% of real critical sections, the accuracy of these inferred critical sections must also be considered.

As a basic starting point, we compare the number of static and dynamic real critical sections with the range of static and dynamic inferred critical sections for shared objects reported by the tool using the range of thresholds specified in Table 2. Table 3 shows the total number of real critical sections statically and dynamically—note that Barnes has a range of 166 to 175 dynamic real critical sections while all other benchmarks have an exact number of real critical sections both statically and dynamically. In the remaining rows, Table 3 shows the minimum and maximum number of inferred critical sections reported when the tool ran over the 112 triplets of thresholds.

The minimum number of inferred static critical sections for shared objects in the original method falls around the number of real static critical sections. We expect it to be the same for all benchmarks except FMM, which has nine static critical sections that do not access heap memory and therefore cannot be found by the tool. The minimum number of inferred static critical sections in the island method is much higher—about two to four times higher than the number of real critical sections. The island method forms more objects since every heap memory reference can become its own object, so it also produces more inferred critical sections. The maximum number of static inferred critical sections for the original method ranged from 5 to 22 times larger than the total number of real static critical sections; unless these extra inferred critical sections overlap significantly with the real critical sections, this suggests a large number of false positives. For the island method, the maximum number of static inferred critical sections ranges from 10 to 45 times the number of real critical sections.

**Table 3: Number of static and dynamic real critical sections and inferred critical sections for shared objects created under island and original methods by benchmark. Boxes with two entries show the minimum and maximum number of critical sections.**

| | **Barnes** | **FMM** | **Ocean Contiguous** | **Ocean Non-Contiguous** | **Water Spatial** | **Water-nSquared** |
|---|---|---|---|---|---|---|
| **Real Static** | **13** | **20** | **26** | **26** | **20** | **22** |
| **Inferred Static** | 20<br>146 | 26<br>93 | 25<br>554 | 20<br>209 | 31<br>170 | 30<br>113 |
| **Inferred Static Island** | 58<br>425 | 36<br>219 | 115<br>1154 | 109<br>405 | 66<br>433 | 65<br>404 |
| **Real Dynamic** | **166**<br>**175** | **181** | **293** | **303** | **94** | **137** |
| **Inferred Dynamic** | 111<br>1983 | 118<br>932 | 237<br>4194 | 162<br>1781 | 135<br>1967 | 178<br>1105 |
| **Inferred Dynamic Island** | 309<br>10112 | 110<br>4898 | 240<br>14442 | 704<br>7922 | 345<br>9025 | 349<br>7472 |

For dynamic inferred critical sections using the original method, only Water Spatial and Water-nSquared have a minimum number of critical sections that is greater than the number of real dynamic critical sections. Since some dynamic real critical sections do not touch heap memory, the tool will not be able to find these sections, so we expect the number of dynamic inferred critical sections to be smaller than the number of dynamic real critical sections. For the island method, the number of dynamic inferred critical sections either falls below the number of real critical sections (FMM and Ocean Contiguous) or is two to four times larger. The maximum number of dynamic inferred critical sections for the original method is 5 to 20 times the number of dynamic real critical sections; for the island method these numbers are 27 to 100 times the number of dynamic real critical sections.

Figure 31 through Figure 42 show how many static critical sections for shared objects were inferred in each benchmark and whether all, some, or none of their instructions were contained in real critical sections. For dynamic critical sections, the results are in Figure 43 through Figure 54.

Like the results for real critical section coverage, a prominent cyclic pattern appears in these graphs. The address difference threshold has the most dramatic effect on the results. When the address difference threshold is eight, the total number of inferred critical sections and the number that have no instructions overlapping with a real critical section skyrocket to their maximums in the island method and plunge to reach their minimums in the original method. The island method creates many more objects than the original method when the address difference threshold is eight, but as that threshold increases, island objects are merged with nearby addresses creating the same objects that the original method creates causing the graphs to look very similar at higher threshold values. Still, some island objects never get merged into larger objects creating a small difference between the island and original methods at higher threshold values. This is why for the remaining values of the address difference threshold the results for the island and original method are about the same. In both methods, as the address difference threshold increases from 16 to 64, the number of inferred critical sections where all instructions cover a real critical section decreases slightly.

The non-object accesses threshold has a small effect on the inferred critical section accuracy. In general, as it gets larger, the inferred critical sections become less accurate. Specifically, as it gets larger, the number of inferred critical sections where every instruction is also in a real critical section decreases, while the number of inferred critical sections where only some instructions are also in a real critical section increases. This is to be expected—the longer we wait to end an inferred critical section, the more likely we will make the inferred critical section too large.

The time threshold for creating objects has almost no effect on the number of inferred critical sections created or their accuracy. As in the statistics for real critical section coverage, this may be the result of testing an insufficient range of threshold values.

There are two ways to count false positives in this data. False positives could be only the inferred critical sections where no instructions cover a real critical section, or it could be the combination of those inferred critical sections with the inferred critical sections where some instructions cover a real critical section. Since the inferred critical sections where some instructions cover a real critical section hint that a real critical section should be present around that area, for now, we will not count them as false positives.

For the original method, the set of thresholds that produces the most inferred critical sections in the "all instructions cover real" category with the fewest false positives is when the address difference threshold is set to eight, the non-object accesses threshold is low, and the time threshold is set to anything. Here, we get very few (as low as two in the static results for Barnes) inferred critical sections that do not touch any part of a real critical section.

For the island method, although the set of thresholds that gives the largest number of inferred critical sections where all instructions cover real critical sections is when the address difference threshold is eight and the non-object accesses threshold is low, this provides the worst false positive rate. The lowest false positive rate occurs at the higher values of the address difference threshold, but as this threshold gets higher, the number of inferred critical sections with all instructions covering real critical sections decreases. Anywhere in this range, the false positive rate is well over 50%.

## *Overall Performance*

Finally, we calculated dynamically the number of instructions that were covered by inferred critical sections grouping them by whether the instruction was part of a real critical section or not. The number of instructions locked unnecessarily (in an inferred critical section but not in a real critical section) is a rough measure of how inferring critical sections influences performance. The results for each benchmark are in Figure 55 through Figure 66 where the number of instructions is on the x-axis and the thresholds are on the y-axis.

As expected, the graphs are very similar to the dynamic results for inferred critical section accuracy. Both methods lock many more instructions than necessary when the address difference threshold is equal to 16, 32 and 64, but for the island method, this trend is even worse for an address difference threshold of 8.

The non-object accesses threshold affects the number of instructions locked more obviously than it affected any of the other results graphs. As the threshold gets larger, there is a slight increase in the number of instructions in inferred critical sections that are also in real critical sections, but there is a more prominent increase in the number of instructions that are not in real critical sections.

As in all other results graphs, the time threshold has almost no effect on the output.

## Static Inferred Critical Section Accuracy

🟥 All Instructions Cover Real 🟩 Some Instructions Cover Real 🟦 No Instructions Cover Real

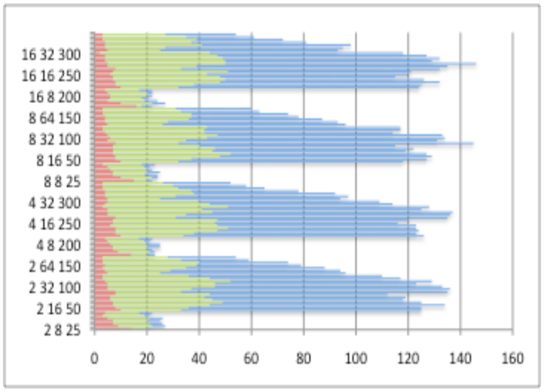


Figure 36: Static inferred critical section accuracy for Barnes.

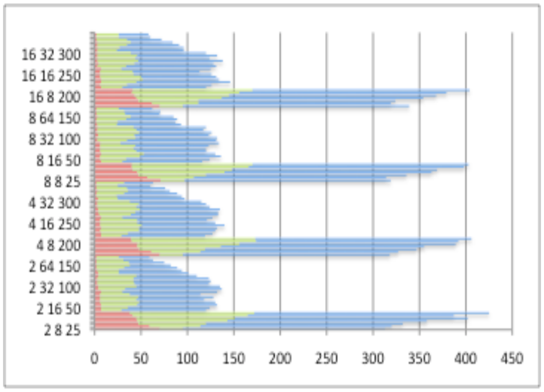


Figure 35: Static inferred critical section accuracy for Barnes using Islands.

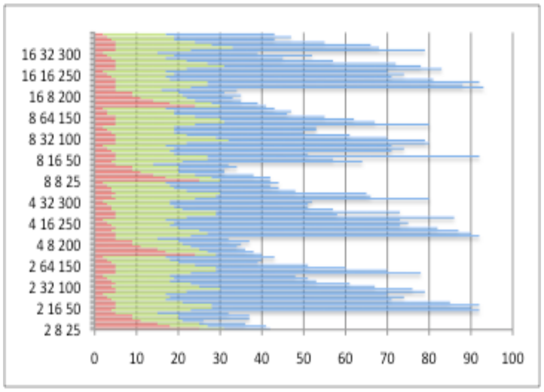


Figure 32: Static inferred critical section accuracy for FMM.

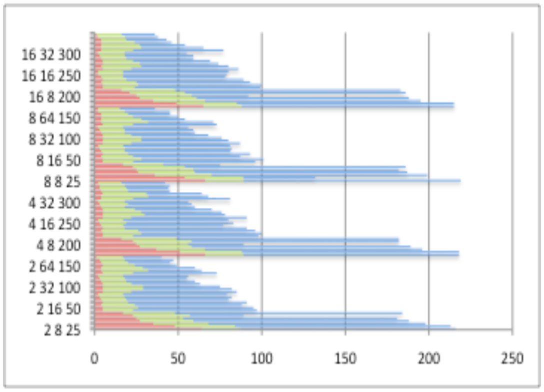


Figure 31: Static inferred critical section accuracy for FMM using Islands.

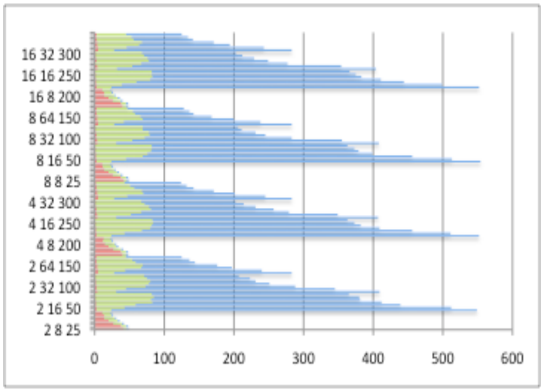


Figure 34: Static inferred critical section accuracy for Ocean Contiguous.

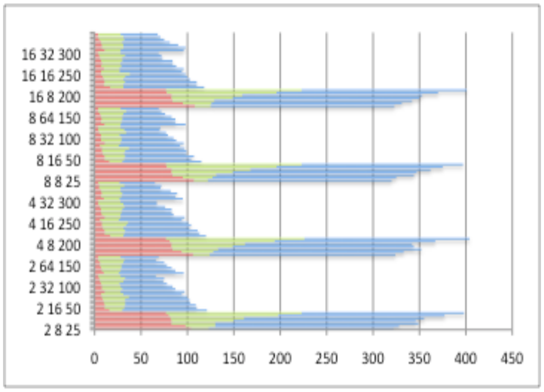


Figure 33: Static inferred critical section accuracy for Ocean Contiguous using Islands.

🟥 All Instructions Cover Real 🟩 Some Instructions Cover Real 🟦 No Instructions Cover Real

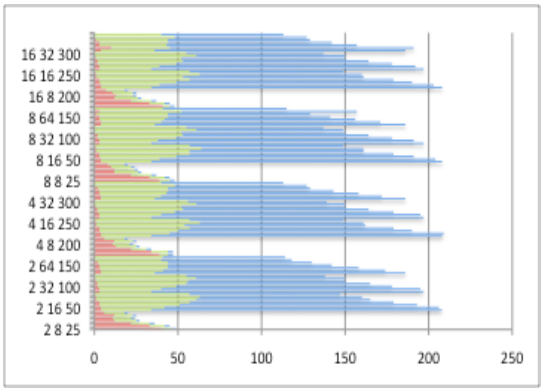


**Figure 42: Static inferred critical section accuracy for Ocean Non-contiguous.**

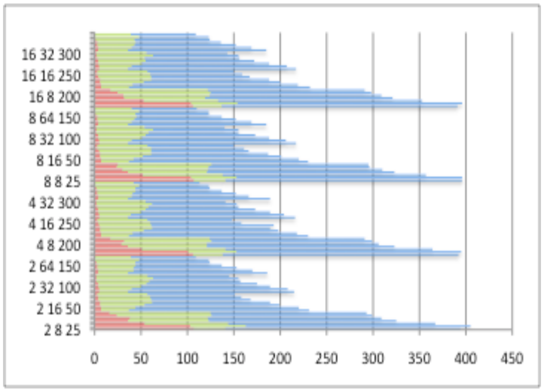


**Figure 41: Static inferred critical section accuracy for Ocean Non-contiguous using Islands.**

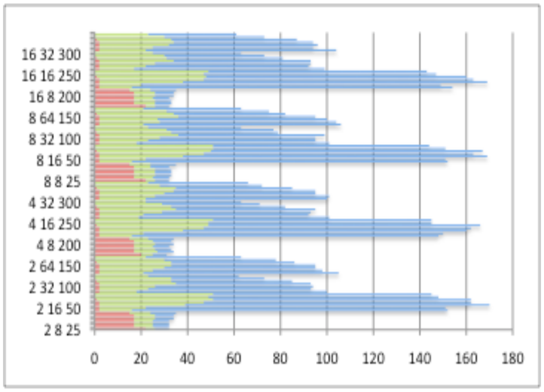


**Figure 40: Static inferred critical section accuracy for Water Spatial.**

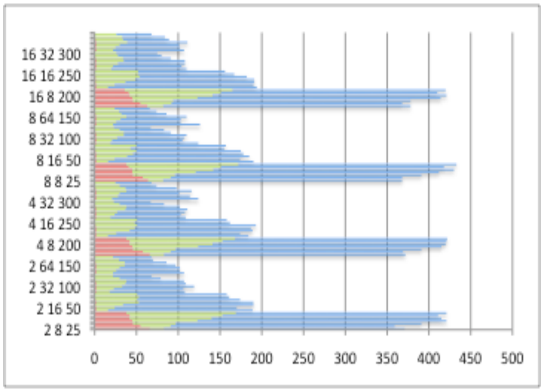


**Figure 39: Static inferred critical section accuracy for Water Spatial using Islands.**

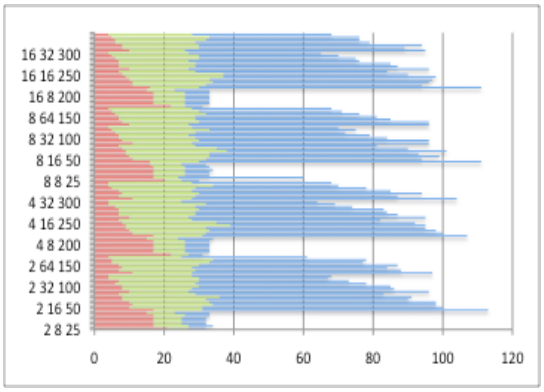


**Figure 38: Static inferred critical section accuracy for Water-nSquared.**

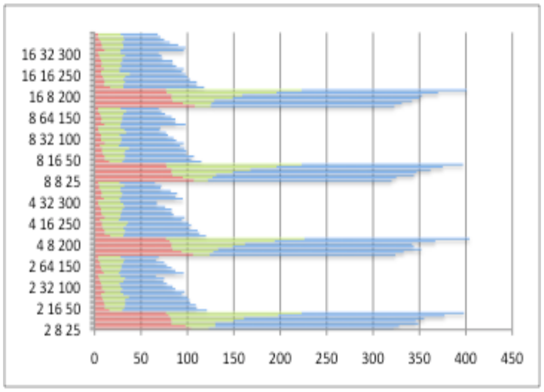


**Figure 37: Static inferred critical section accuracy for Water-nSquared using Islands.**

## Dynamic Inferred Critical Section Accuracy

🟥 All Instructions Cover Real 🟩 Some Instructions Cover Real 🟦 No Instructions Cover Real

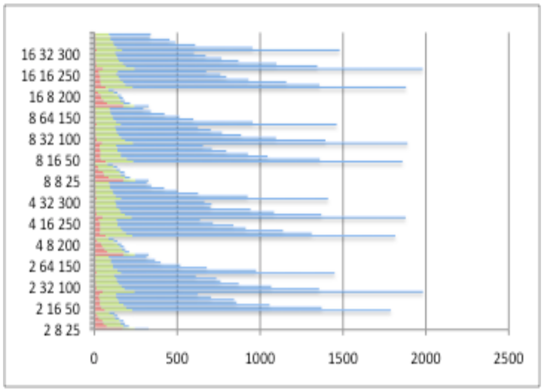


**Figure 48: Dynamic inferred critical section accuracy for Barnes.**

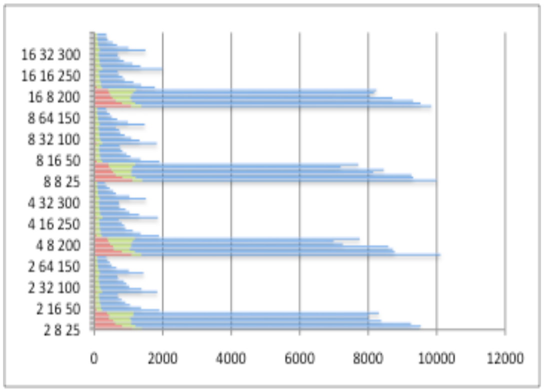


**Figure 47: Dynamic inferred critical section accuracy for Barnes using Islands.**

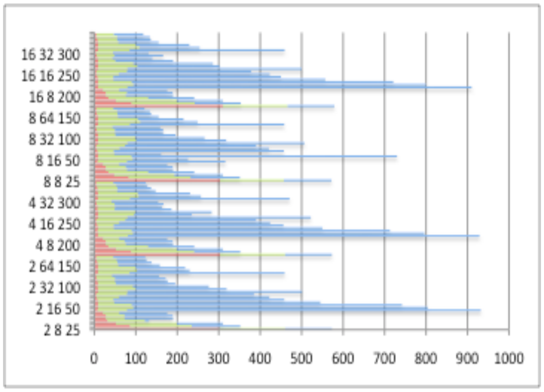


**Figure 46: Dynamic inferred critical section accuracy for FMM.**

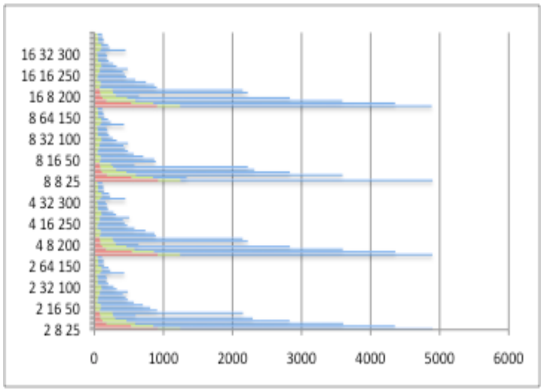


**Figure 45: Dynamic inferred critical section accuracy for FMM using Islands.**

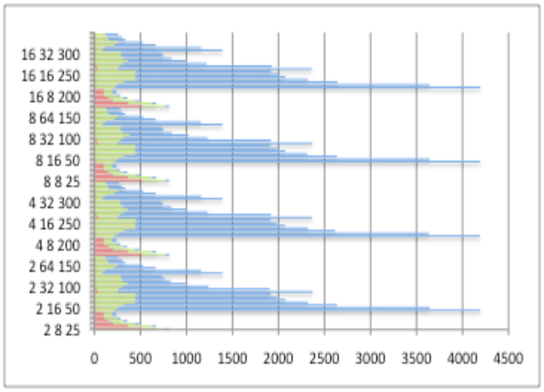


**Figure 44: Dynamic inferred critical section accuracy for Ocean Contiguous.**

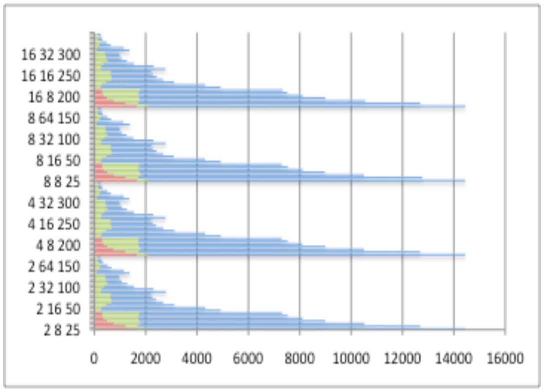


**Figure 43: Dynamic inferred critical section accuracy for Ocean Contiguous using Islands.**

🟥 All Instructions Cover Real 🟩 Some Instructions Cover Real 🟦 No Instructions Cover Real

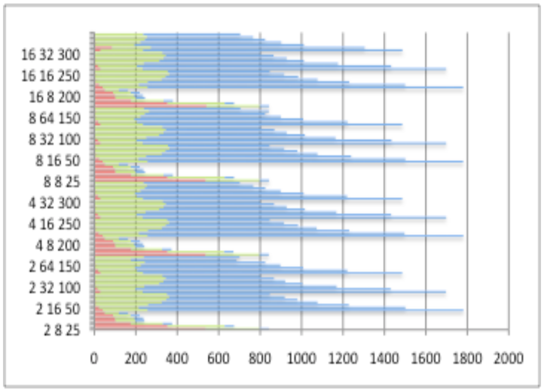


**Figure 54: Dynamic inferred critical section accuracy for Ocean Non-contiguous.**

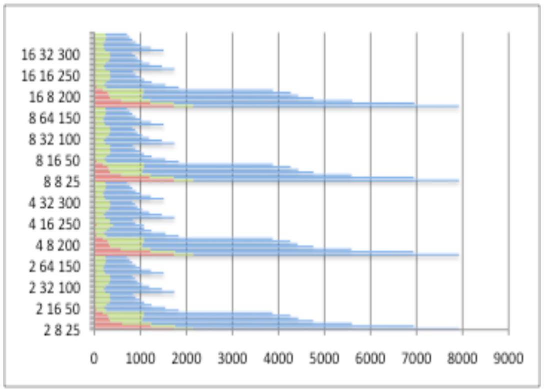


**Figure 53: Dynamic inferred critical section accuracy for Ocean Non-contiguous using Islands.**

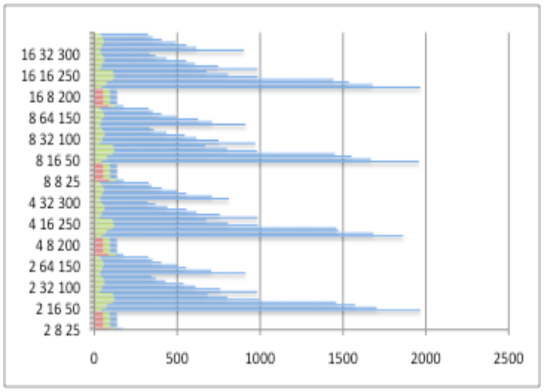


**Figure 52: Dynamic inferred critical section accuracy for Water Spatial.**

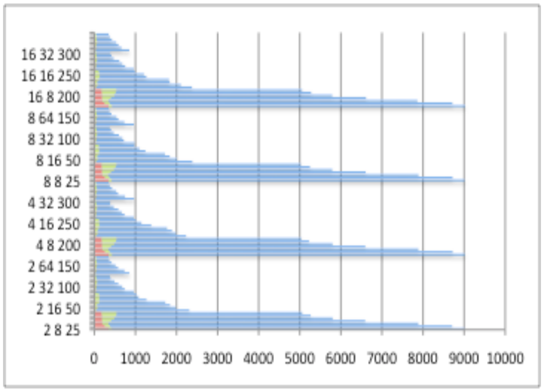


**Figure 51: Dynamic inferred critical section accuracy for Water Spatial using Islands.**

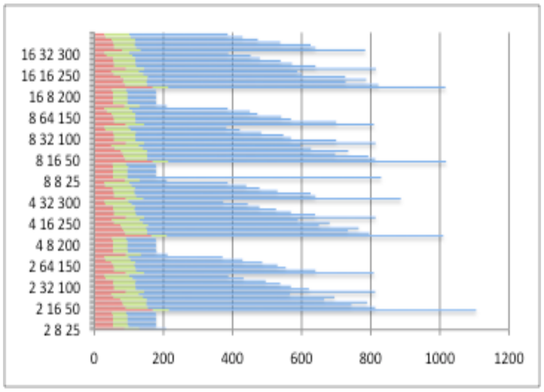


**Figure 50: Dynamic inferred critical section accuracy for Water-nSquared.**

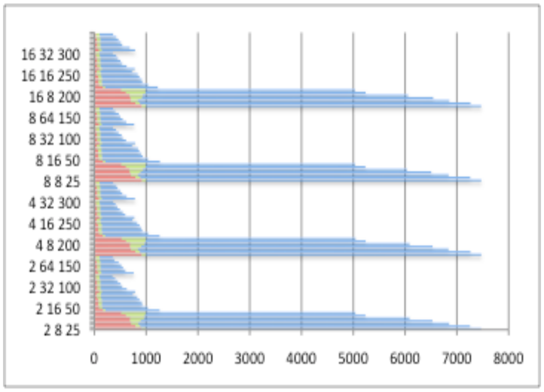


**Figure 49: Dynamic inferred critical section accuracy for Water-nSquared using Islands.**

## Inferred Critical Sections Dynamic Instruction Coverage

🟦 Number of instructions in at least one real critical section
🟥 Number of Instructions in no real critcial sections

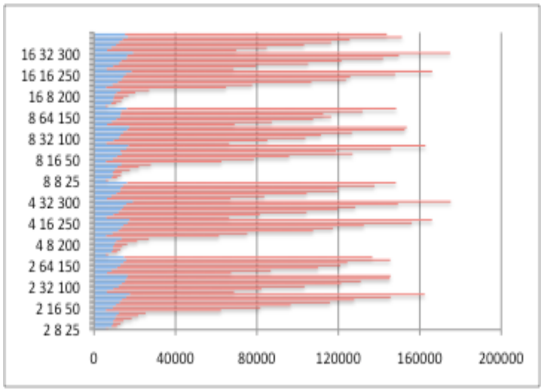


Figure 59: Inferred critical sections' instructions by real overlap for Barnes.

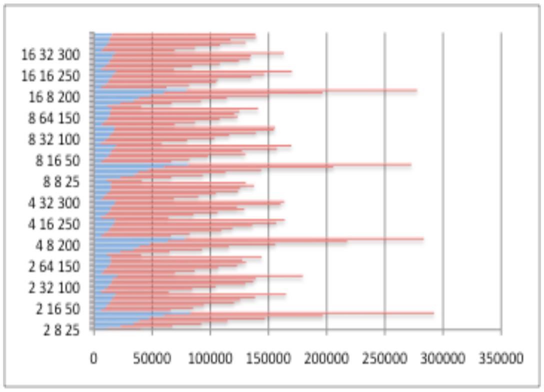


Figure 60: Inferred critical sections' instructions by real overlap for Barnes using Islands.

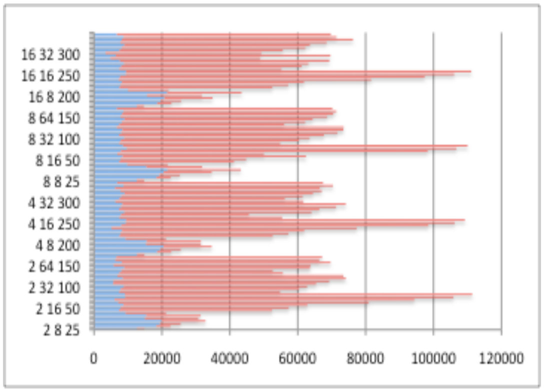


Figure 58: Inferred critical sections' instructions by real overlap for FMM.

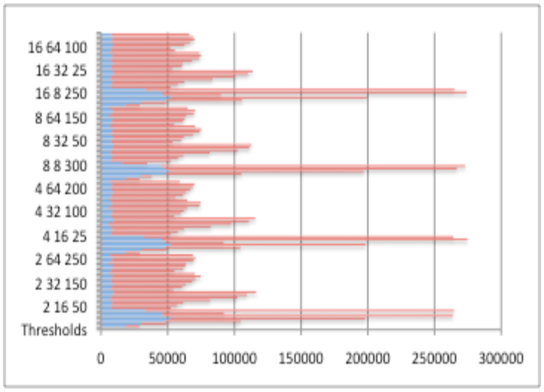


Figure 57: Inferred critical sections' instructions by real overlap for FMM using Islands.

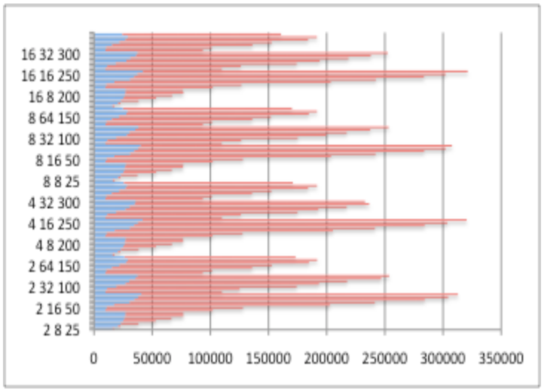


Figure 55: Inferred critical sections' instructions by real overlap for Ocean Contiguous.

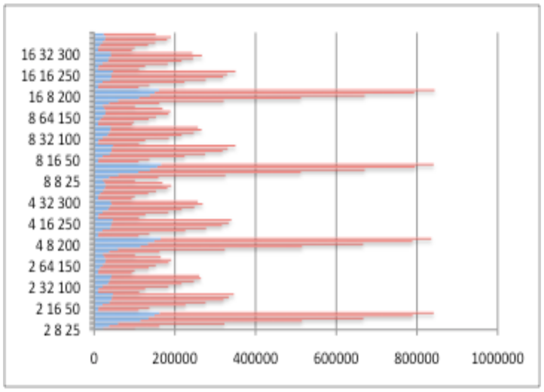
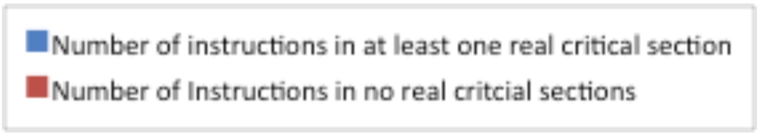


Figure 56: Inferred critical sections' instructions by real overlap for Ocean Contiguous using Islands.



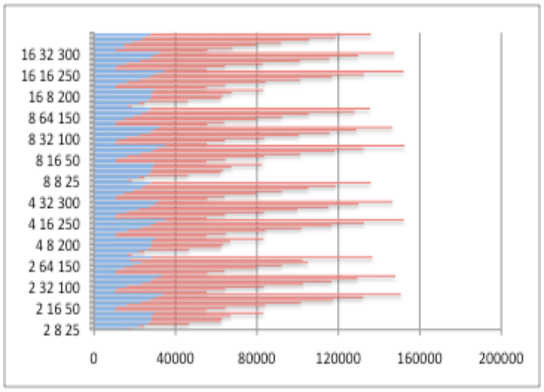


Figure 66: Inferred critical sections' instructions by real overlap for Ocean Non-contiguous.

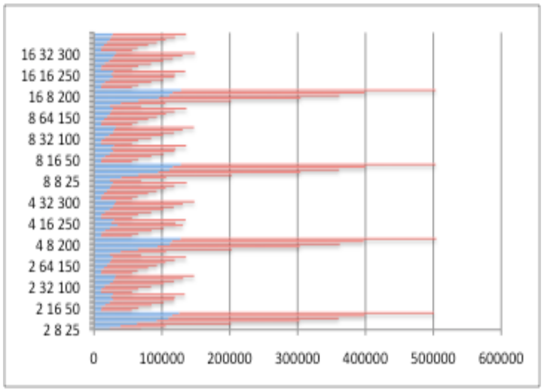


Figure 65: Inferred critical sections' instructions by real overlap for Ocean Non-contiguous using Islands.

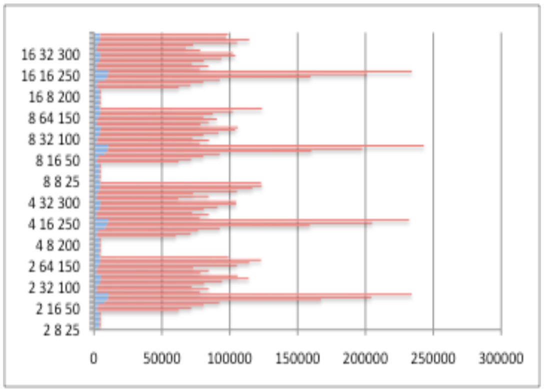


Figure 64: Inferred critical sections' instructions by real overlap for Water Spatial.

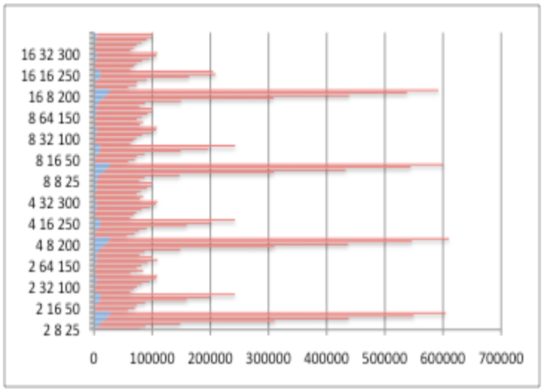


Figure 63: Inferred critical sections' instructions by real overlap for Water Spatial using Islands.

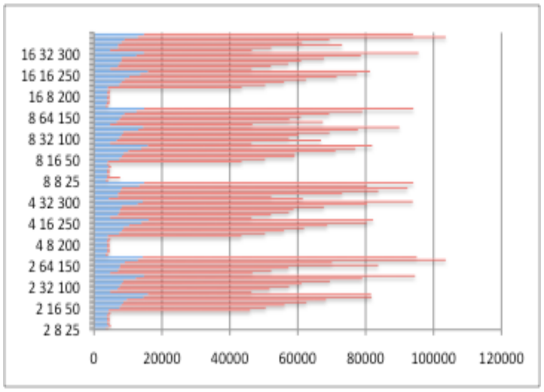


Figure 62: Inferred critical sections' instructions by real overlap for Water-nSquared.

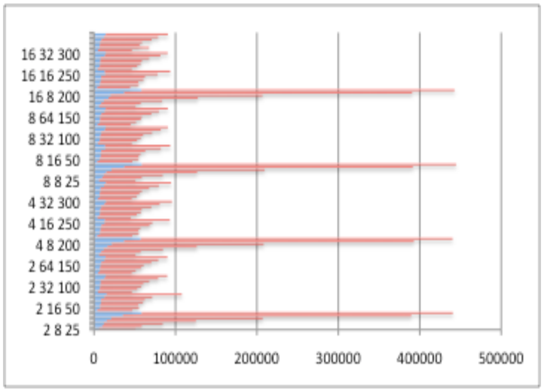


Figure 61: Inferred critical sections' instructions by real overlap for Water-nSquared using Islands.

## *Examples*

Now that we have seen the overall results, we investigate some specific examples of how well the tool covered real critical sections from Water-nSquared. We begin by comparing results from the tool using an address difference threshold of 8, time difference threshold of 8, and non-object accesses threshold of 200—which will be noted (8, 8, 200)—with the results using the address difference of 32, time difference of 8, and non-object accesses threshold of 200—which will be noted (8, 32, 200). As Figure 15 and Figure 16 show, the number of static real critical sections that are completely covered for thresholds (8, 8, 200) is 4 using the original method and 10 using the island method, while for thresholds (8, 32, 200) that number is 17 for both methods.

### Example: Covered Real Critical Section with Excess

For thresholds (8, 8, 200), the real critical section in Code Segment 5 from the function *MDMAIN* in mdmain.c (line 47 of mdmani.C) is fully covered using the regular method by inferred critical sections that do not end prior to the end of this critical section. This critical section can be found in lines 114 to 166 of mdmain.c and is the result of expanding a BARRIER macro on line 47 of mdmain.C. It is dynamically executed four times: once for each thread as it reaches the barrier. As a result, the number of memory instructions in each dynamic execution varies: one thread executes only 19 memory instructions in this critical section while the other three execute 63, 64, and 74 memory instructions.

```
unsigned long   Error, Cycle;
long            Cancel, Temp;

Error = pthread_mutex_lock(&(gl->start).mutex);
if (Error != 0) {
      printf("Error while trying to get lock in barrier.\n");
      exit(-1);
}

Cycle = (gl->start).cycle;
if (++(gl->start).counter != (NumProcs)) {
      pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &Cancel);
      while (Cycle == (gl->start).cycle) {
            Error = pthread_cond_wait(&(gl->start).cv, &
                                          (gl->start).mutex);
            if (Error != 0) {
                  break;
            }
      }
      pthread_setcancelstate(Cancel, &Temp);
} else {
      (gl->start).cycle = !(gl->start).cycle;
      (gl->start).counter = 0;
      Error = pthread_cond_broadcast(&(gl->start).cv);
}
pthread_mutex_unlock(&(gl->start).mutex);
```

**Code Segment 5: Critical section in mdmain.c lines 114 to 166, which is the result of expanding a BARRIER macro in mdmain.C line 47.**

For each dynamic execution of this real critical section, the tool infers three critical sections. One inferred critical section includes all of the memory instructions in the real critical section, while two additional inferred critical sections begin after the real critical section starts and end after the real critical section ends.

Each of the inferred critical sections that completely cover the dynamic executions of this real critical section begins at the same memory instruction as the real critical section, but ends after the real critical section begins. The object associated with all of the inferred critical sections is 16 bytes long and begins at *gl->start*. In fact, *gl->start* points to a structure of size 16 that can be found in Code Segment 6. Since the tool ignores all instructions within *pthread_mutex_lock*, the first reference to this object—and therefore, the start of the inferred critical sections—is in the line `Cycle = (gl->start).cycle;`.

```
struct {
      pthread_mutex_t mutex;
      pthread_cond_t  cv;
      unsigned long   counter;
      unsigned long   cycle;
} (start);
```

**Code Segment 6: Structure *start*: the object that associated with the inferred critical sections that fully cover the real critical section in mdmain.c lines 114 to 166.**

Table 4 compares the number of memory instructions in the inferred critical sections with the number in the dynamic real critical sections. For the three shortest real critical sections, their inferred critical sections end at the same memory instruction 55 memory instructions after the real critical section's end. For the longest real critical section, the final instruction in the inferred critical section is different and occurs 30 memory instructions after the real critical section ends. In both situations the ending instruction is inside the next barrier (line 79 in mdmain.C and beginning at line 219 in mdmain.c). The code for this barrier is identical to the code in Code Segment 5. For the shortest three real critical sections, their fully covering inferred critical sections end during the function *pthread_cond_wait* with a final reference to the *gl->start* object. For the longest real critical section, its fully covering inferred critical section ends during the function *pthread_cond_broadcast*. Both of these functions take a pointer to the condition variable *cv* that is in the *gl->start* object as an argument.

**Table 4: Lengths of dynamic executions of the critical section from mdmain.c lines 114 to 166 compared with the length of the inferred critical sections that fully cover each dynamic execution.**

| | | | | |
|---|---|---|---|---|
| Real Critical Section Length | 19 | 63 | 64 | 74 |
| Inferred Critical Section Length | 74 | 118 | 119 | 104 |

This example shows where an inferred critical section correctly points out the object that should be protected, but incorrectly locks that object beyond the necessary time.  Depending on the way the tool is used, this inferred critical section could be useful by pointing out an object and region that the programmer may want to lock or it could be detrimental if it were used to automatically lock down a running program. In the latter case, two barriers could be treated as one causing the program to attempt to do computations on uninitialized or outdated data.

## Example: Uncovered Real Critical Section Covered with Island Method

The example in Code Segment 7 shows a real critical section from the function *INTERF* in interf.c (lines 160-162 and in interf.C lines 145-147) that is left uncovered using the original method, but is covered using the island method for the same set of thresholds (8, 8, 200).  This real critical section references a local value *LVIR* and the pointer *VIR*, which is passed into the function as an argument.  Every time the function is called, this argument points to a global variable *VIR*, a `double` that is dynamically allocated at the start of execution as part of Water-nSquared's *GlobalMemory* structure. There are two memory instructions captured by our tool in this critical section: one that adds the local value *LVIR* and **VIR* and one that stores **VIR* back to memory.

```
{pthread_mutex_lock(&(gl->InterfVirLock));};
*VIR = *VIR + LVIR;
{pthread_mutex_unlock(&(gl->InterfVirLock));};
```

**Code Segment 7: Real critical section in lines 160 to 162 of interf.c (corresponding to lines 145-147 of interf.C).**

This real critical section is dynamically executed twice by each thread for a total of eight times in our trials. It is never covered using the original method for these thresholds because despite being dynamically allocated, *VIR* is not referenced near any other memory references so it does not become part of an object. When the address difference threshold is increased from 8 to 32, *VIR* becomes part of an object with a total of 328 bytes in the original method. Obviously, this is much larger than necessary, but once *VIR* is part of an object, this real critical section gets covered exactly by an inferred critical section.

With the same (8, 8, 200) thresholds, the island method makes *VIR* into its own 8-byte object. The inferred critical section associated with *VIR*'s object perfectly covers this real critical section. Since more objects are formed in the island method, more inferred critical sections are formed also. Between one and nine inferred critical sections fully cover the dynamic instances of this real critical section. For every dynamic instance, at least one of the inferred critical sections is a perfect match the dynamic real critical section; the other inferred critical sections all start before this code region.

While a perfectly fitting inferred critical section is the goal of this tool, in this example, we see it comes with a trade-off: too many extra, meaningless inferred critical sections. However, the island method, with between one and nine inferred critical sections covering this code region, strongly suggests that this very small, two memory instruction critical section should be covered.

## Example: Real Critical Section in a Loop

The only real critical section in kineti.c is inside a *for*-loop. The code for this critical section and the *for*-loop containing it is in Code Segment 8. Each thread executes this critical section three times. The locked variable, *SUM*, is an array of three `doubles` (8-byte quantities). The critical section itself has two memory instructions that are captured by our tool.

```
/* loop over the three directions */
for (dir = XDIR; dir <= ZDIR; dir++) {
      S=0.0;
      /* loop over the molecules */
      for (mol = StartMol[ProcID]; mol < StartMol[ProcID+1];
           mol++) {
            double *tempptr = VAR[mol].F[VEL][dir];
            S += ( tempptr[H1] * tempptr[H1] +
                  tempptr[H2] * tempptr[H2] ) * HMAS
                  + (tempptr[O] * tempptr[O]) * OMAS;
      }
      {pthread_mutex_lock(&(gl->KinetiSumLock));};
      SUM[dir]+=S;
      {pthread_mutex_unlock(&(gl->KinetiSumLock));};
} /* for */
```

**Code Segment 8: Real critical section in kineti.c lines 47 to 60 (kineti.C lines 32 to 45).**

For the original method using the thresholds (8, 8, 200), no inferred critical sections cover the first execution in every thread, but a single inferred critical section for each thread covers the last two dynamic executions suggesting that between the executions of the real critical section the tool possibly learns the object to create the critical section. Unfortunately, this is not what happens. The object associated with this single inferred critical section is not related to the locked variable *SUM*; instead the object is related to the *pthreads* implementation and accessed for the first and last times inside the function *__pthread_mutex_unlock_usercnt*, a function called in *pthread_mutex_unlock*. This indicates that the inferred critical section that covers the last two dynamic executions of the critical section begins during the first call to *pthread_mutex_unlock*, the call that ends the first dynamic execution of the real critical section, and runs until after the last dynamic execution of the real critical section.

Using the island method and the same thresholds, the tool correctly identifies the three `doubles` in the *SUM* array as independent objects. The inferred critical sections it creates for these objects start correctly at the first memory instruction of the real critical section, but they end well after the real critical section ends—they are between 40 and 124 memory instructions long instead of two instructions. When the first index of *SUM* is accessed in the first dynamic instance of the real critical section, the associated inferred critical section continues to be active through the next two dynamic instances of the real critical section; the same thing happens after the second index of *SUM* is accessed. In fact, these inferred critical sections all end at the next reference to *SUM*, which occurs in mdmain.c on line 463 (mdmain.C line 118) after the call to *KINETI* returns and all the threads move through a barrier. This suggests that the non-object accesses threshold is too large, causing the tool to wait too long to determine the end of an inferred critical section, so that it includes memory references that are unrelated to the true real critical section. If our Pintool detected loop iteration boundaries like AtomTracker [20], this information could be used to force the critical section to end at the boundary even though the threshold has not been met. Future work could include adding this feature.

Using the island method and reducing the non-object accesses threshold to 25 fixes the problem of the excessively large critical sections. When the other thresholds are held constant (8, 8, 25), the tool correctly finds this real critical section each time it is accessed. Additionally, the number of inferred critical sections found for each dynamic execution of this real critical section is down from 9 to 14 for the non-object accesses threshold at 200 to between 1 and 3.

This example shows the power of the non-object accesses threshold for getting more accurate inferred critical sections. If a program is expected to have small real critical sections, a smaller non-object accesses threshold should be used. If the real critical sections are larger with accesses to the locked object spread out in the critical section, a larger non-object accesses threshold will work better.

# Chapter 5: Discussion

Chapter 4: Results shows that the memory usage pattern of a program can be used to learn about a program's real critical sections; however, the effectiveness of this result depends on how it is used. Would this information best be incorporated into a tool that automatically locks down a program while it runs or a tool that advises a programmer of potential bugs? We discuss a few potential uses below. First, though, we look at improvements to our analysis that would provide more informative results.

## *Improving the Results*

Our analysis incorrectly included instructions that were inside *pthread* library functions, because Pin was unable to completely identify both the dynamic beginning and end of all pthread library functions. For example, Pin could not find the return statement for the *pthread_mutex_unlock* function, so we were unable to remove instructions occurring inside *pthread_mutex_unlock* from analysis. Rather than relying on Pin to find the ends of *pthread* library functions, a better system of recognizing instructions inside libraries and ignoring them should be implemented. Additionally, we should ensure that all *pthread* library functions are removed from analysis and not just *pthread_mutex_lock* and *pthread_mutex_unlock*. The benchmark programs also used *pthread_cond_braodcast* and *pthread_setcancelstate*, which we should have recognized and eliminated.

Our analysis was limited by the fact that we only had enough computing power to analyze heap memory references for inferring objects. The Pintool computations relied on C++'s Standard Template Library Containers to organize the large amounts of data that had to be sorted and searched. Since not all features of the C++ containers were used, an implementation that used more efficient vectors, lists, and maps potentially could handle including non-heap memory references in the object inference analysis.

Finally, the way we reported our results was an artifact of how we collected coverage information while processing. Consequently, it is difficult to answer the question, "How many inferred critical sections perfectly match real critical sections?" While we have seen in the examples provided at the end of Chapter 4: Results that several inferred critical sections do perfectly match real critical sections, at this time, we do not have precise statistics to judge how well the tool performs in this regard.

## *Potential Uses*

The two most common uses for a tool like this would be to incorporate it into an advisor program that informs the programmer of code regions that might need to be critical sections and to incorporate the results into a second program that automatically inserts critical sections into the code. Because a human must interact with the results, an advisor program must have a low false positive (unnecessary inferred critical sections) rate; on the other hand, with an automatic program false positives can be tolerated better because they only impact performance. Inferred critical sections that do not accurately match real critical sections can cause incorrect behavior for an automatic program, but for the advisor program, a programmer might be able find the correct critical section provided the incorrect critical section.

Because the tool produces a total number of inferred static critical sections that ranges from 2 to 20 times the number of static real critical sections, using the tool as an advisor program would require careful selection of thresholds to keep the false positives to a manageable number for the programmer. Note that the thresholds that produce the fewest inferred static critical sections for these benchmarks (when the address difference threshold is 8) are the same thresholds that perform the worst in covering real critical sections. An advisor program would have to trade off manageability for the programmer with accuracy.

A program that automatically inserts locks into the benchmark program would need to have the best coverage of real critical sections possible. This means using thresholds where the address difference threshold is greater than 8. Note that having best coverage possible does not mean that the program will function correctly. False positives affect performance by unnecessarily locking regions that are not critical sections. The original and island methods have about the same high false positive rates when the address difference threshold is greater than 8. Fortunately, increasing both the address difference threshold and the non-object access threshold decreases the false positive rate while increasing the real critical section coverage. Finally, the automatic tool would have to avoid creating deadlock when adding locks to the program. This will be discussed further in the next section.

# Chapter 6: Future Work

This study presents only the very first results in a promising area of atomicity research. Here we suggest several additions to the tool and to testing that could provide better results. Finally, we discuss the challenges of building a complete tool to use the results of this study.

## *Analyzing New Benchmarks*

We evaluated our tool's performance on SPLASH-2 scientific computing benchmarks only. The next step for this research should determine if the results hold across other types of programs. AVIO [16], Atom-Aid [17], and AtomTracker [20] evaluate their performance using real-world server and client applications such as MySQL, Apache, and Mozilla, and our tool should be evaluated on these also.

Although we ignored locking functions (as much as possible) in our benchmarks, our results were limited to matching inferred critical sections with real critical sections in otherwise correctly functioning code. By expanding our study to real-world applications, we can also incorporate the real-world bugs found in these applications to determine if our tool would correctly find critical sections in incorrect code. As the "Learning from Mistakes" study showed, adding locks is not always the best fix for a bug, so if a bug is fixed by reordering code or changing the design of a data structure, will our tool still detect these buggy regions as critical sections [15]?

## *Improving Object and Critical Section Inference*

Our object and critical section inference is based solely on the memory locality and temporal proximity of memory accesses, where locality is defined by thresholds. Incorporating other tools into our critical section and object inference could improve the results.

Our tool could benefit from recognizing and modifying object and critical section inference in loops. AtomTracker [20] works on the assumption that critical sections either contain an entire loop or never cross loop iteration boundaries. Incorporating this idea into our tool could provide for better object and critical section inference. For example, our tool could rely on loop detection to identify the individual objects within an array when a loop iterates over an array of objects rather than relying on thresholds to identify the individual objects.

LoopProf [18] is one of several loop detection techniques that could be used to add this feature. LoopProf was written as a Pintool, making it easy to incorporate into our tool. In addition to providing loop boundaries, LoopProf provides other profile information about loops including parent and child loops and the nesting depth of a loop [18]. LoopProf could be incorporated as a preprocessing step for our tool.

## *Creating a Completely Automated Tool*

As addressed in the section Potential Uses above, given the large number of unnecessary critical sections formed by our tool, a tool that automatically inserts locks around inferred critical sections may be the best use of this tool. Our tool already reports the object that must be locked and the critical sections for that object.

The work that remains is to create a lock for each object and ensure that the program does not deadlock.

In “Lock Inference for Atomic Sections,” Hicks et al. [11] address adding mutexes for objects and appropriately locking regions already known to be atomic sections. They are able to reduce the total number of mutexes needed by determining when two or more mutexes are always held together. They avoid deadlock by creating a total ordering of all mutexes and only acquiring locks in order [11].

Incorporating the work of Hicks et al. into our tool would allow us to build a tool that automatically inserted locks into the program ensuring that the program never deadlocked.

Finally, to fully compare our tool with AtomTracker [20], we could run the new lock-instrumented program counting the number of atomicity violations that have been prevented.

# Appendices

**Barnes Input**

Nbody = 16
Seed = 123
Dtime = 0.025
Eps = 0.05
Tol = 1.0
Fcells = 2.0
Fleaves = 5.0
Tstop = 0.075
Dtout = 0.25
NPROC = 4

**FMM Input**

Cluster type = two cluster
Distribution type = plummer
Number of Particles = 15
Precision = 1e-2
Number of Processors = 4
Number of Time Steps = 5
Duration of Time Step = .025
Softening Parameter = 0.0

**Ocean Contiguous Input**

```
-n6 -p2 -e1e-07 -r2000000000 -t28800
```

**Ocean Non-Contiguous Input**

```
-n6 -p2 -e1e-07 -r2000000000 -t28800
```

**Water-nSquared Input**

TSTEP = 1.5e-15
NMOL = 8
NSTEP = 1
NORDER = 6
NSAVE = 0
NRST = 0
NPRINT = 1
NFMC = 0
Num Procs = 4

CUTOFF = 6.212752

**Water-Spatial Input**

TSTEP = 1.5e-15
NMOL = 8
NSTEP = 1
NORDER = 6
NSAVE = 0
NRST = 0
NPRINT = 1
NFMC = 0
Num Procs = 4
CUTOFF = 0

# Glossary

**Address difference threshold**: The maximum number of bytes that can be between two effective addresses and still merge the addresses into a single object.

**Atomicity**: Operations within an atomic region occur without interruption of operations on another processing element, so to other processing elements they appear to happen at once.

**Atomicity violation**: A concurrency bug in which a critical section's final state is influenced by another processing element. See page 10.

**Critical section**: A region of code that accesses a shared resource and must do so exclusively.

**Distributed-memory model**: In this parallel processing model, each processing element has exclusive access to its data and cannot access the data of other processing elements.

**Island method**: Every address accessed is its own object and can be merged into larger objects.

**Lock**: Operation of gaining exclusive access to data or resources by locking a mutex. Also, a lock can be an object that can be locked or unlocked to gain mutually exclusive access to data or resources.

**MISD**: Multiple Instruction-Single Data organization of processing elements in which each processing element executes a different instruction but on the same datum.

**MIMD**: Multiple Instruction-Multiple Data organization of processing elements in which each processing element executes a different instruction on a different set of data.

**Mutex**: Object that is locked and unlocked to guarantee a thread's mutually exclusive access to data or resources.

**Non-object accesses threshold:** The number of memory accesses to addresses that are not in the current critical section's object before determining the critical section has ended.

**Object**: A contiguous region of memory built by merging together memory accesses that are nearby in time and space.

**Order violation**: A concurrency bug where the programmer assumes that two threads will execute in a particular order but does not enforce this order. See page 11.

**Original method**: Objects are created from merging accesses. Single accesses that have no nearby accesses (within the address difference threshold and time threshold) do not become objects.

**Parallel computing**: Performing two or more computations simultaneously.

**Processing element**: Any unit that can perform a computation.

**Shared-memory model:** In this parallel processing model, all processors have access to a shared bank of memory and the memory of every other processor.

**SIMD**: Single Instruction-Multiple Data organization of processing elements. All processing elements execute the same instruction simultaneously but on different data.

**SISD**: Single Instruction-Single Data organization of processing elements. This is serial computing where a single instruction is executed on a single datum at a time.

**Serializable**: Parallel events that could be executed in some serial order to get the same final state.

**SPMD**: Single Program-Multiple Data organization of processing elements. Each processing element executes the same program, but processing elements can execute different parts of the program at a time. Each processing element operates on different data.

**Thread**: Stream of instructions running inside a larger process that can be independently scheduled by the operating system. Threads have their own stack, registers, and private data, but have access to all global data.

**Time threshold:** The maximum number of memory accesses that can be between accesses to two effective addresses and still merge the addresses into a single object.

**Transaction**: A group of instructions executed optimistically atomically. Transactions must be serializable.

**Unlock**: Releasing mutually exclusive access to data or resources.

# Bibliography

[1] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *IEEE International Symposium on Workload Characterization, 2008*, p. 47-56, September 2008.

[2] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Conistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, p. 278-289, 2007.

[3] S. Cherem, T. Chilimbi, S. Gulwani. Inferring Locks for Atomic Sectios. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, p. 304-315, 2008.

[4] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace for Multithreaded Object-Oriented Programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, p. 258-269, June 2002.

[5] F. Darema. "The SPMD Model: Past, Present and Future." In Y. Cotronis and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, Berlin/Heidelberg, 2001.

[6] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. In *IEEE Computational Science & Engineering*, Volume 5 Issue1, p. 46-55, January 1998.

[7] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. In *Communications of the ACM*, 39(7), p. 84-90, July 1996.

[8] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The Notions of Consistency and Predicate Locks in a Database System. In *Communications of the ACM*, p. 624-633, November 1976.

[9] M. Flynn. Some Computer Organizations and Their Effectiveness. In *IEEE Transactions On Computers*, Vol. C-21, No. 9, p. 948-960, September 1972.

[10] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. In *Proceedings*

*of the 31st Annual International Symposium on Computer Architecture*, p. 102, 2004.

[11] M. Hicks, J. Foster, P. Pratikakis. Lock Inference for Atomic Sections. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[12] M. Herlihy and J.E.B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, p. 289-300, 1993.

[13] A. Lai and B. Falsafi. Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, p. 139-148, June 2000.

[14] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. Popa, and Y. Zhou. MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, p. 103-116, 2007.

[15] S. Lu, S. Park, E. Seu, and Y. Zhou. Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, p. 329-339, 2008.

[16] S. Lu, J. Tucek, F. Qin, Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 37-48, 2006.

[17] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *Proceedings of the 35th International Symposium on Computer Architecture*, p. 277-288, June 21-25, 2008.

[18] The Modified SPLASH-2 Home Page. 2007. CAPSL. Univeristy of Delaware. <http://www.capsl.udel.edu/splash/Download.html>

[19] T. Moseley, D. Grunwald, D. Connors, R. Ramanujam, V. Tovinkere, and R. Peri. LoopProf: Dynamic Techniques for Loop Detection and Profiling. In *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.

[20] A. Muzahid, N. Otsuki, and J. Torrellas. AtomTracker: A Comprehensive Approach to Atomic Region Inference and Violation Detection. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, p. 287-297, 2010.

[21] R. Netzer, B. Miller. What are Race Conditions? Some Issues and Formalizations. In *ACM Letters on Programming Languages and Systems (LOPLAS)*, p. 74-88, March 1992.

[22] Pin – A Dynamic Binary Instrumentation Tool. <www.pintool.org>

[23] POSIX Threads Programming. Barney, Blaise. Lawrence Livermore Laboratories. <https://computing.llnl.gov/tutorials/pthreads/>.

[24] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, p. 24–36, June 1995.

[25] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J. E. Smith, and M. Valero. Implementing Kilo-Instruction Multiprocessors. In *International Conference on Pervasive Services*, p. 325-336, 2005.

[26] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-wait-free Multiprocessors. In *International Symposium on Computer Architecture*, p. 266-277, 2007.