Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs

[Extended Abstract] *

Eli Pozniansky
Computer Science Department
Technion—Israel Institute of Technology
kollega@cs.technion.ac.il

Assaf Schuster
Computer Science Department
Technion—Israel Institute of Technology
assaf@cs.technion.ac.il

ABSTRACT

Data race detection is highly essential for debugging multithreaded programs and assuring their correctness. Nevertheless, there is no single universal technique capable of handling the task efficiently, since the data race detection problem is computationally hard in the general case. Thus, to approximate the possible races in a program, all currently available tools take different "short-cuts", such as using strong assumptions on the program structure or applying various heuristics. When applied to some general case program, however, they usually result in excessive false alarms or in a large number of undetected races.

Another major drawback of many currently available tools is that they are restricted, for performance reasons, to detection units of fixed size. Thus, they all suffer from the same problem—choosing a small unit might result in missing some of the data races, while choosing a large one might lead to false detection.

In this paper we present a novel testing tool, called MultiRace, which combines improved versions of Djit and Lockset—two very powerful on-the-fly algorithms for dynamic detection of apparent data races. Both extended algorithms detect races in multithreaded programs that may execute on weak consistency systems, and may use two-way as well as global synchronization primitives.

By employing novel technologies, MultiRace adjusts its detection to the native granularity of objects and variables in the program under examination. In order to monitor all accesses to each of the shared locations, MultiRace instruments the C++ source code of the program. It lets the user fine-tune the detection process, but otherwise is completely automatic and transparent.

This paper describes the algorithms employed in MultiRace, discusses some of its implementation issues, and proposes several optimizations to it. The paper shows that the overheads imposed by MultiRace are often much smaller (orders of magnitude) than those obtained by other existing dynamic techniques.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Concurrency, multiprocessing/ multiprogramming/ multitasking, synchronization, threads*; D.3.4 [**Programming Languages**]: Processors—*Memory management, Preprocessors, Run-time environments*

General Terms

Algorithms, Performance

Keywords

Data race, Concurrency, Multithreading, Instrumentation, Synchronization

1. INTRODUCTION

Multithreading is a common programming paradigm that is well-suited for multiprocessor environments. The obvious advantages of multithreading over single threaded programming are parallelism and improved performance. However, multithreading also introduces the problem of data races.

A *data race* occurs when two or more threads concurrently access a shared location without synchronization, and at least one of the accesses is for writing. Such a situation is usually considered to be an error (a.k.a. a *bug*). In most cases it is undesirable, as it might lead to unpredictable results and incorrect program execution. Data races usually stem from errors made by programmers who fail to place synchronization correctly in the program.

Unfortunately, the problem of deciding whether a given program contains potential data races is computationally hard. Researchers define *feasible data races* as races that are based on the possible behavior of the program (i.e., the semantics of the program computation) [15]. These are the real races that might happen in any specific execution of a program. According to [13], the problem of exactly locating feasible data races is NP-hard in the general case.

Since data races are usually a result of improper synchronization that does not prevent concurrency of accesses, researchers also define *apparent data races* [15]. These are

*The full version of the paper can be found at `www.cs.technion.ac.il/~assaf/publications/MultiRace.pdf`

PPoPP'03 June 11–13, 2003, San Diego, California, USA.

approximations of feasible data races, based on the behavior of the explicit synchronization only; they are defined in the context of a specific program execution. Apparent data races are simpler to locate than feasible data races, but they are also less accurate than the latter. It was proved that apparent races exist if and only if at least one feasible race exists somewhere in the execution. Yet according to the same paper, the problem of exhaustively locating all apparent data races is still NP-hard.

Methods for detecting apparent data races use one of two main approaches—either *static* or *dynamic* detection. Sometimes a combination of these approaches is used for additional efficiency and accuracy. The static methods [3, 7] perform a compile-time analysis of the program's source code. Their advantage is that they check the program globally—they are able to warn about any data race that might occur in an execution of the program. In addition, since they operate directly on the code, they can perform the data race detection in granularity of objects as declared by the structures and classes of the program. Their main drawback is that they are too conservative in the general case—they can neither know nor understand the real semantics of the program. Hence, for every modern full-scale programming language, static detection methods always result in excessive false alarms, which confuse the programmer and mask the real data races.

The dynamic methods, which are further divided into *postmortem* and *on-the-fly* techniques, use some tracing mechanism to detect whether a particular execution of the program actually exhibited data races. The postmortem techniques [2, 14] collect some information about the order of synchronization and computation events during the execution, and create a trace of the run. When the execution terminates, they analyze this trace and warn about possible data races. In contrast, on-the-fly methods [5, 10, 16, 17] buffer partial trace information in memory, analyze it, and detect data races as they occur. Thus, some of these methods are also capable of pinpointing the exact locations of the instructions involved in a detected race.

The advantage of the dynamic methods over the static is that they detect only those apparent data races that actually occurred during real executions. Their main drawback, however, is that they check the program locally by considering only one specific execution path of the program each time. If the program takes another path when, for example, another input is supplied, other data races might pop up. Hence, in order to detect all data races, all possible execution paths should be checked. This, however, is not practical in most cases. Thus, the dynamic check should not be restricted to program testing time, but should also be activated each time the program executes, or in case a problem is suspected.

The above discussion brings up the issue of overhead. Dynamic techniques that are known in the literature, especially those that work on-the-fly, impose high slowdowns on the program under examination. Typically, this overhead is in the order of hundreds to thousands of percentage points, thus precluding the activation of the detection in production mode. Commonly, there exists a tradeoff between the runtime overhead and the accuracy of the detection, namely, lower overhead will result in missing more real data races, or in reporting more false alarms.

Finally, the issue of *detection granularity* (i.e., the size of the memory block/unit on which the detection is actually performed) is also of utmost importance: a small unit results in missed data races and higher overheads, whereas a large one causes false detection. In modern object-oriented programming languages, objects are the natural choice for becoming the units on which the detection should be applied: they usually contain data that is strongly related, and should be protected as a whole. For this reason the locking granularity in modern languages, such as Java (as realized in the *monitor* APIs), is applied in granularity of objects[1]. We remark that the average object size is known to be very small (about 20 bytes for Java objects), hence the average detection accuracy on objects is relatively good. Unfortunately, many currently known dynamic methods are limited to detection granularity of fixed size (e.g., fixed number of bytes, like double-word), and thus cannot perform the detection in object-size granularity. There are two main reasons for this limitation: performance considerations and inability to correctly detect object boundaries.

In this paper we present a novel testing tool called MultiRace, which combines two very powerful techniques for on-the-fly detection of apparent data races. The first technique is a revised version of the Djit algorithm [10], called Djit$^+$. The algorithm is based on Lamport's *happened-before* partial order relation [11], and it is capable of efficiently detecting apparent races as they occur. The second technique is an improved variant of the Lockset algorithm [17], which warns about the shared locations for which the *locking discipline*, a policy common among programmers, is violated during the program's run. Both Djit$^+$ and improved Lockset detect data races in programs that execute on weakly ordered systems, and use global and two-way synchronization primitives: barriers and locks. In fact, they can be easily extended for use with programming models that require other common synchronization primitives.

The benefit of combining these algorithms in one tool is that they complement each other in a way that the one compensates for the shortcomings of the other. For example, Lockset usually reports the same set of locking discipline violations under different thread scheduling. Yet such violations can be just false alarms that do not necessarily lead to occurrences of feasible data races. On the other hand, the Djit$^+$ algorithm detects only those data races that actually occurred during the execution. Yet it is very sensitive to differences in thread interleavings. Thus, it can miss some or even all of the data races due to some especially unfortunate execution. Combining both algorithms in one tool and applying them to the same program execution at the same time makes the detection of data races much more powerful.

MultiRace makes use of several novel technologies. By exploiting a unique configuration of memory mappings, called *views*, and the technique of *pointer swizzling* [4, 9], MultiRace detects data races in granularity of variables and objects in the program, rather than in fixed-size units. To the best of our knowledge, it is the first entirely transparent on-the-fly framework for multithreaded environments that is capable of doing so. MultiRace carries out this task with the help of automatic and transparent source code instrumentation. In this approach, the code of the tested program, written in C++, is pre-processed, modified, and recompiled, such that calls to logging and detection mechanisms are injected

[1]Although locking in the lower granularity of code blocks is also allowed in Java, it is rarely used in practice.

in places where accesses to shared locations are performed.

Finally, by logging only a portion of all the accesses to shared locations, MultiRace imposes an overhead that is smaller by orders of magnitude than those imposed by other currently available tools. This fact makes it an even more attractive tool for on-the-fly data race detection in multithreaded environments. The slowdowns measured for six common benchmark applications are low enough to make its use practical, whereas the average slowdowns reported in previous on-the-fly works, like [5, 16, 17], are much higher for similar kinds of applications.

Because of its transparency, relatively low overhead, powerful detection algorithms, and ability to match the detection granularity to that of the objects used in the program, MultiRace is unparalleled by any of the known dynamic detection techniques for multithreaded environments.

The rest of this paper is organized as follows. Section 2 describes the assumptions of our memory and synchronization models, gives the definition of a data race, presents the detection algorithms used in our system, and discusses the benefits of combining them in the same framework. Section 3 gives the main ideas behind the logging mechanism implemented in MultiRace: it describes the memory organization and the management of views. Section 4 explains our notion of the variable-size detection granularity. Section 5 gives highlights of the transparent instrumentation process, which enables the access logging. Section 6 describes how MultiRace reports data races in the tested program. Section 7 suggests several optimizations of MultiRace, and Section 8 presents the obtained overheads. Section 9 surveys related works. We give our conclusions in Section 10.

2. DETECTION ALGORITHMS

2.1 Memory and Synchronization Models

The technique proposed in this paper assumes a model of some multithreaded environment, in which, in addition to local accesses, the threads read from and write to shared locations of the process within which they reside. Shared locations are assumed to be global and static objects, as well as all allocations from the heap.

In order to prevent concurrent accesses and avoid data races, the threads use *synchronization primitives* on shared *synchronization objects*. We follow the basic observation that a synchronization object S can be *released* by one set of threads that reach a certain point in their execution and can then be *acquired* by another set of threads. For example, an `unlock(S)` operation releases S and a corresponding `lock(S)` operation acquires it; a global `barrier(S)` operation causes all threads to release S as they reach the barrier, and only then causes them to re-acquire S. To simplify our discussion, we deal with set of programs that employ only locks and barriers. Other more complicated synchronization primitives, such as semaphores, monitors, etc., can be viewed in similar terms.

The most common and easy to understand model for shared memory is *sequential consistency*. This model implies the existence of an agreed-among-all-threads global order $\mathcal{R}$ on all shared memory accesses in the program execution. Under this order, the reads of every shared location v always return the most recently written value to v. The problem with sequential consistency is that it is very restrictive and, hence, it limits the employment of many optimizations that would be possible otherwise. Thus, modern memory models weaken the restrictions of the memory behavior.

In our work we assume some *weakly ordered* system that follows the *data-race-free-1* shared memory model definition, first presented in [1]. This model only requires that the program should appear sequentially consistent in the total absence of data races. This also means that in the presence of data races there is not necessarily an inter-thread global order of variable modifications.

2.2 Definition of a Data Race

We base our definition of data races on the *happened-before* relation first suggested by Lamport for message passing systems [11]. This definition also resembles those presented in [2, 10]; it allows recognition of apparent data races that pop up in some specific program execution.

DEFINITION 1. *The* happens-before *partial order, denoted* $\overset{hb}{\rightarrow}$*, is defined for all computational events (reads, writes) and synchronization events (releases, acquires) that happen in a specific execution. We say that* $\alpha \overset{hb}{\rightarrow} \beta$ *if: (1)* α *and* β *are any two events performed by the same thread, with* α *preceding* β *in the program order, or (2)* α *is a release and* β *is its corresponding acquire, both operating on the same synchronization object* S*, or (3)* $\alpha \overset{hb}{\rightarrow} \gamma$ *and* $\gamma \overset{hb}{\rightarrow} \beta$*.*

DEFINITION 2. *We say that two events* α *and* β *are synchronized if either* $\alpha \overset{hb}{\rightarrow} \beta$ *or* $\beta \overset{hb}{\rightarrow} \alpha$*. We say that* α *and* β *are concurrent if they are not synchronized.*

DEFINITION 3. *We say that there exists an apparent data race between two accesses to the same shared location,* α *and* β*, if* α *and* β *are executed by distinct threads, they are not synchronized (i.e., they are concurrent), and at least one of them is for writing.*

2.3 Djit$^+$

In order to detect data races, we use an algorithm called Djit$^+$, which is a revised version of the earlier Djit algorithm [10]. The main disadvantages of the original Djit were the assumption of an underlying sequentially consistent system and the ability to detect only the very first data race in an execution. In contrast, Djit$^+$ can correctly operate on weakly ordered systems and still detect a greater number of races as they occur in the program's execution.

Djit$^+$ relies on a formal framework called *vector time frames*, which is based on Mattern's virtual time vector time-stamps [12]. The algorithm also assumes the existence of some logging mechanism (to be described later), capable of dynamically recording all accesses to each of the shared memory locations. The general idea of the algorithm is to log every shared access and to check whether it "happens-before" prior accesses to same location.

2.3.1 Realizing the $\overset{hb}{\rightarrow}$ Relation

The execution of each thread is logically split into a sequence of *time frames*. A new time frame starts each time the thread performs a *release* operation. The time frames are numbered in a monotonically increasing order.

We assume that the maximum number of threads is known and stored in a constant called *maxThreads*. Each thread t maintains a vector of time frames, denoted $st_t[.]$, having *maxThreads* entries. For the sake of simplicity we assume

that the IDs of the threads are positive integers in the range of [0,*maxThreads*-1]. For each index u, the entry $st_t[u]$ stores the latest local time frame of thread u, whose release operation is "known" to thread t. During the execution, the vectors of the threads are maintained in the following way: (1) if t performs a release of some synchronization object S, it increments the value of $st_t[t]$, and by this enters a new time frame; (2) if u acquires a synchronization object S previously released by t, then each entry in u's vector is updated to hold the maximum between its old value and the corresponding value in t's vector at the moment of the release. (Clearly, in order to correctly update the entries of u's vector, it is necessary to store the vector of t after the *release* in t and before the corresponding *acquire* in u. This is done by assigning a similar vector of time frames, denoted $st_S[.]$, to each synchronization object S. The actual way in which the updates of thread vectors are performed, is described in Subsection 2.3.3.)

In the full paper we prove that the $\stackrel{hb}{\rightarrow}$ relation can be verified from the vectors of the time frames defined above. Thus, to detect data races on-the-fly it is sufficient to check the time frame vector of each newly logged access with the time frame vectors of all previously logged accesses. Indeed, this is the main idea used in our Djit$^+$ algorithm.

2.3.2 Reducing the Number of Checks

An algorithm that logs and checks all the accesses with all the previously logged accesses, even if correct, obviously imposes high overhead on the system. A couple of simple observations allow us to restrict the logging and checking to only a portion of the set of all accesses, thus reducing the total overhead while still maintaining correctness.

The fist observation is that in order to determine whether some shared location participated in any data races during an execution, it is sufficient to log only the first read and the first write accesses to this location in each time frame. Similarly, it is sufficient to check for races only between those accesses to the same shared location, which are the first in their respective time frames.

The second observation is that if there is no data race between two accesses, α and β, then there can be no data races between α and accesses that appear prior to β in the program order. Developing this observation we find that it is sufficient to check the current write access to a shared location v against the last time frame in each of the other threads which recently read from v and the last time frame in each of the other threads which recently wrote to v. For the current read access, it is sufficient to check it against the last time frame in each of the other threads that wrote to v.

The above distinction between reads and writes arises from the fact that a read access constitutes only races with writes, while a write access constitutes races with reads as well as with writes. Therefore, the set of accesses with which a read access should be checked for a race is a subset of the set of accesses with which a write access should be checked.

2.3.3 The Detection Protocol

In order to actually implement the detection protocol described in previous subsections, each thread t and each synchronization object S hold a vector of time frames, as described beforehand. In addition, every shared location v holds, for each thread t, two parameters—the last time frame of t in which it wrote to v, and the last time frame in which it read from v. This information is called the *access history of* v and is denoted $aw_v[t]$ and $ar_v[t]$ respectively. Thread t first updates its entry, and only then reads the entries related to other threads. Accesses to the access history, $aw_v[t]$ and $ar_v[t]$, are atomic, meaning that the threads always read a consistent state of the values that were previously written. Figure 1 shows the full Djit$^+$ protocol.

Upon initialization :

1. Each initializing thread t fills its vector of time frames with ones—$\forall i : st_t[i] \leftarrow 1$.
2. The access history of each shared location v is filled with zeros (since no thread has accessed it yet)—$\forall i : ar_v[i] \leftarrow 0, aw_v[i] \leftarrow 0$.
3. The vector of each synchronization object S is filled with zeros—$\forall i : st_S[i] \leftarrow 0$.

Upon a release of synchronization object S :

1. The issuing thread t starts a new time frame. Therefore, it increments the entry corresponding to t in t's vector—$st_t[t] \leftarrow st_t[t] + 1$.
2. Each entry in S's vector is updated to hold the maximum between the current value and that of t's vector—$\forall i : st_S[i] \leftarrow max(st_t[i], st_S[i])$.

Upon an acquire of synchronization object S :

1. The issuing thread t updates each entry in its vector to hold the maximum between its current value and that of S's vector—$\forall i : st_t[i] \leftarrow max(st_t[i], st_S[i])$.

Upon a first access to a shared location v in a time frame or a first write to v in a time frame :

1. The issuing thread t updates the relevant entry in the history of v. If the access is a read, it performs $ar_v[t] \leftarrow st_t[t]$. Otherwise, it performs $aw_v[t] \leftarrow st_t[t]$.
2. If the access is a read, thread t checks whether there exists another thread u which also wrote to v, such that $aw_v[u] \geq st_t[u]$. In other words, t checks whether it knows only about a release that preceded the write in u, and if so reports a data race.

   If the access is a write, thread t checks whether there exists another thread u, such that $aw_v[u] \geq st_t[u]$ or $ar_v[u] \geq st_t[u]$.

Figure 1: The full Djit$^+$ protocol

Note that absence of announced races only ensures that the given execution is data race free. It still does not imply that the entire program is free of races. If, on the other hand, races are found, the programmer can be notified and supplied with the exact locations of the racing instructions in the code.

2.4 Lockset

To perform even more efficient detection of data races we also use a refined and optimized version of the Lockset algorithm, first presented in [17]. Our implementation takes

Upon initialization:

1. For each v, $C(v)$ is initialized to the set of all possible locks.

Upon an access to v by thread t:

1. $lh \leftarrow locks\_held(t)$.
2. If the access is a read, then $lh \leftarrow lh \cup \{readers\_lock\}$.
3. $C(v) \leftarrow C(v) \cap lh$.
4. If $C(v) = \emptyset$, then a race warning is issued.

Figure 2: The Lockset Algorithm

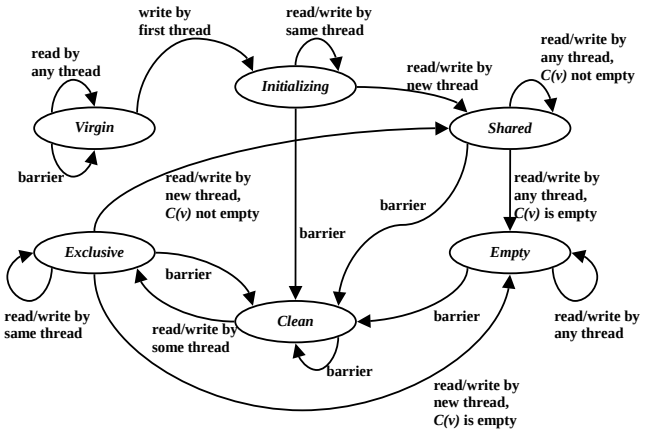

Figure 3: The state transition diagram used in our extended Lockset

advantage of the time frames idea, which, as was the case for Djit$^+$, makes it possible to decrease the required number of checks. In addition, we extend the basic Lockset algorithm to use barriers, so that it can be integrated with Djit$^+$ within the same framework.

2.4.1 The Basic Algorithm

The basic Lockset algorithm detects violations of a *locking discipline*. A simple, yet common locking discipline is to require that each shared location be protected by the same lock on each access to it. Clearly, such a policy ensures the total absence of data races in a program. Yet a violation of the discipline is not always a bug and does not necessarily lead to a data race. Therefore, the main drawback of the algorithm is that it might result in an excessive number of false alarms, which hide the real data races. Nonetheless, this technique was actually implemented in a full scale testing tool called Eraser [17], and it was shown to provide very important and powerful results.

For the sake of clarity, we next describe in general terms the idea behind the algorithm. For each shared location v, its candidate set, denoted $C(v)$, is defined to be the set of all locks that have consistently protected v on each access to it in the execution so far. For each thread t, $locks\_held(t)$ holds at any given moment the set of all locks acquired by t. The algorithm itself, also called *lockset refinement*, is depicted in Figure 2.

Note that there is a distinction between reads and writes in the depicted algorithm that does not exist in original Lockset. On each read access we simulate the acquisition of an additional "fake lock", denoted *readers_lock*. In this way, multiple reads in different threads that are not protected by any locks will not produce false alarms. (In the original Lockset algorithm, another more complicated technique that achieves the same result was used.) Clearly this does not prevent the reads from executing concurrently. However, the first write to v permanently removes *readers_lock* from $C(v)$. Thus, in order that $C(v)$ does not become empty, another "real lock" is required to consistently protect v.

2.4.2 Reducing the Number of Checks

One of the disadvantages faced by the inventors of Lockset was the overhead incurred due to the monitoring of all accesses to each of the shared locations. When we compared the Djit$^+$ algorithm with the lockset refinement described above, we noticed that it is possible to significantly reduce the overhead of Lockset by recording only the first accesses in each of the time frames, in the same way as was done in Djit$^+$. The reason for this is quite obvious. Consider two accesses, α and β, to some shared location v, such that they are both in the same thread, α precedes β in the program order, and they occur during the same time frame. In such a case, there are no *unlock* or *barrier* operations between α and β, yet there can appear any number of *lock* operations. Thus the set of locks held during access α to v is a subset of the set of locks held during access β to v. Therefore, Lockset does not obtain any additional information by checking accesses that are not first in their respective time frames.

2.4.3 Supporting Barriers

The original Lockset, as well as the refined version discussed so far, do not take advantage of the barrier synchronization primitive. In order to correctly support barriers, we observe that the definition of a barrier does not allow a pair of racing accesses, such that one access happens before the barrier and the other one after it. This suggests that after reaching a barrier, the candidate sets of all shared locations must be re-initialized by setting them to hold all possible locks. Then the detection should be restarted.

For this purpose, we use a technique similar to the one used in the original Lockset. For each shared location we employ an extended state transition diagram. This diagram is depicted in Figure 3. It controls the refinement and the maintenance of the candidate sets, as well as the announcement of race warnings.

The main differences between our diagram and the one used in the original Lockset, are the *Clean* and *Exclusive* states. In addition, we assume that a shared variable is initialized not only when it is first accessed by a second thread (as in the original Lockset), but also when the thread that first wrote to it reaches a barrier. Thus, if all threads reach a barrier, every shared location that has already been initialized, changes its state to *Clean*. When the *Clean* state is entered, $C(v)$ is modified to hold the set of all possible locks, as in the initialization phase of the algorithm (Figure 2). When location v is first accessed after the barrier, it reaches the *Exclusive* state. At this point, v is assumed to be already initialized, and hence $C(v)$ is refined each time v is accessed by the same thread. The race warning is issued only if v is accessed by an additional thread and $C(v)$ is

empty (either prior to the access or due to the refinement caused by it).

Note that our support for barriers is added almost transparently to the original algorithm. It is easy to see that it does not produce any new false alarms and it does not miss any possible races other than those described in [17]. Clearly, our refined version can be used to check programs that employ barriers only. In contrast, the original version of Lockset will produce an overwhelming number of false alarms in this case, even if these programs are correct and data race free.

2.5 Benefits of Combining Lockset and Djit$^+$

Most of the overhead in implementing Lockset and Djit$^+$ is in the logging mechanism, shared by both algorithms. Thus, it is tempting to combine them into the same tool, enabling more powerful detection of data races. The resulting benefits of applying both algorithms to the same execution at the same time are as follows:

- Lockset alone cannot distinguish between real races and false alarms. In contrast, Djit$^+$ detects only those apparent races that actually occurred. The combination of the algorithms supplies the programmer with additional vital information as to which shared locations are actually raced and which are not.

- In contrast to Djit$^+$, the Lockset algorithm was found to be quite insensitive to differences in thread interleavings, and it was shown to provide a certain kind of global information about the raced shared locations in a program.

- Since every data race is also a violation of the locking discipline, for many types of programs it can be said that Lockset and Djit$^+$ detect respectively a superset and a subset of all the raced shared locations in the execution. Therefore, it can be concluded that if Lockset did not produce any warnings in some execution, then there is a high probability that Djit$^+$ will not locate any additional races in further executions. Figure 4 demonstrates this observation.

- The number of checks performed by Djit$^+$ can be reduced using the additional information obtained from Lockset. If the current access to some shared location v does not empty the candidate set $C(v)$ (i.e., v is still consistently protected), then we can be sure that this access does not form a data race with earlier accesses to v. Thus, Djit$^+$ should not perform any checks of the access history of v if $C(v)$ is not already empty.

3. IMPLEMENTATION OF THE LOGGING MECHANISM

In the following sections we describe MultiRace—the actual framework for implementing the logging mechanism and the data race detection algorithms. In this section we give a description of the memory organization and management that enable the access logging mechanism. This description is quite general. It does not assume any specific programming language, but only requires some common characteristics of the underlying operating system.

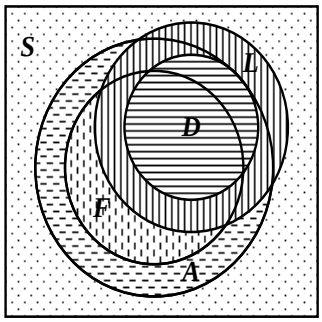

Figure 4: The set of all shared locations in some given program P is represented by region S. The set of shared locations in P that are participating in feasible data races is represented by region F. The set of shared locations in P that are participating in apparent data races is represented by region A, which is a superset of F. The set of shared locations for which Lockset detects violations of the locking discipline in some specific execution E_P of P is represented by region L. The set of shared locations that Djit$^+$ reports as participating in data races in E_P is represented by region D, which is always a subset of both A and L. We remark that for many types of programs (e.g., such that are not *completely nondeterministic* [6]), A becomes a subset of L.

3.1 View Swizzling Approach

As was discussed earlier, our logging mechanism needs to record only the first accesses (reads and writes) to shared locations in each of the time frames. Techniques suitable for this task were introduced in [9] and [4], which presented the concept of *views*. According to this concept, a physical memory page can be viewed from several virtual pages, called *views*, each having its own protection. Each object that resides on the corresponding physical page can be accessed through each of the different views. This attribute helps to distinguish between read and write accesses to the shared objects. In addition, this enables the realization of the variable-size detection unit, and thus avoids the fixed-size granularity problem usually faced by other data race detection tools. The concept of views and the memory layout imposed by it are depicted in Figure 5.

We refer to the shared locations that are accessed using the views approach as *minipages*. Each minipage is associated with the information essential for the data race detection algorithms. In addition, each minipage can be referenced through one of the three views: `NoAccess`, `ReadOnly` or `ReadWrite`. Accessing a minipage through the wrong view (e.g., writing through a `ReadOnly` view) generates a page fault, which can be caught by the operating system. Clearly, the `NoAccess` view will catch each access to it, the `ReadOnly` view will catch only writes, and the `ReadWrite` view will not generate any page faults.

Modern operating systems allow a user handler function to be provided for different kinds of software and hardware exceptions. In the case of a page fault, the handler is supplied with the faulted memory address, the faulting instruction, the type of page fault (read or write), and the states of the machine registers. Once this information has been

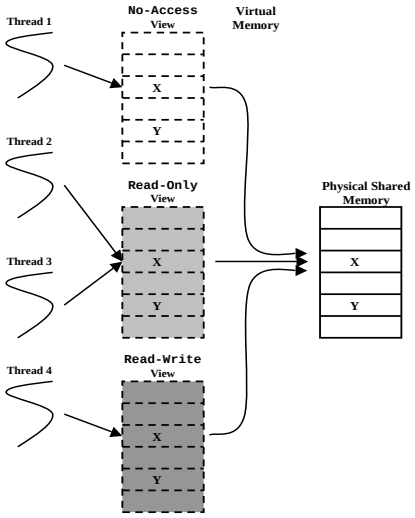

Figure 5: The memory layout and the depiction of views. Note that different threads can access X through different views, thus getting different protections for it.

obtained, appropriate action can be taken: the faulted minipage can be located, its view tested and modified, and the race detection mechanisms invoked.

Recall that implementing the detection algorithms requires logging only the first accesses to shared locations in each time frame. The idea for the logging mechanism is therefore straightforward. We use a technique called *pointer swizzling*, also employed in [4]. When some thread is initialized, or after it performs a *release* operation indicating the beginning of a new time frame, it points to all minipages through the `NoAccess` view. This process of swizzling the views to `NoAccess` is called *invalidation*. If the first access to some minipage in the current time frame is a read, a page fault occurs and the thread modifies its view on this minipage to `ReadOnly`. In this way, subsequent reads do not generate any faults, but a later write in the same time frame produces a write fault. If this happens, the view is changed to `ReadWrite`. If, on the other hand, the first access in a time frame is a write, the view is moved directly to `ReadWrite`, so later accesses do not produce any faults. It is easy to see that such a protocol correctly distinguishes between those accesses that are important for the detection mechanisms and those that are not.

3.2 Memory Layout and Memory Allocations

In order to achieve the memory layout depicted in Figure 5, the system, when activated, enters an initialization phase. This phase precedes the initialization of global variables and the execution of any line of code in the tested program. In this start-up phase, a physical memory object, large enough to satisfy all future program's memory requests, is allocated. This object is the *shared memory area* recognized by our system for the purpose of access logging and data race detection. Then, the `NoAccess`, `ReadOnly` and `ReadWrite` views are mapped on that memory object in the virtual memory.

In order to intercept the memory allocation requests, all allocation routines and operators are overridden. In this way, the program's allocation requests are always satisfied from the shared memory object. Each memory allocation request creates a minipage. The minipage's starting address is always returned to the programmer through the `ReadWrite` view, so that constructors, for example, are invoked without being faulted.

3.3 Swizzling Pointers

Suppose that our program contains a pointer that holds an address of some shared object through the `NoAccess` view. As long as this program pointer is only manipulated (i.e., assigned to other pointers or incremented by some offset, etc.), no race detection or swizzling should be performed. The moment the pointer is dereferenced and the pointed object is actually accessed, a page fault occurs, and the race detection mechanisms are activated. The view on the object is then swizzled to the `ReadOnly` or the `ReadWrite` view.

The problem with this scenario is that the program pointer itself is not changed. In fact, there is no simple way of changing the pointer without at least knowing its address, which is not supplied to the page fault handler. Moreover, if there are some additional pointers to the same shared object or to offsets inside it, they all have to be swizzled as well. In the RTL system, presented in [4], the suggested technique is to force each pointer that refers to a shared memory area to be recognized by that area. In this way, all pointers referencing the area can be swizzled at once. This method, besides being very restrictive and slow (there can be any number of pointers referencing the same area), assumes user assistance in identifying these pointers.

In contrast, our technique is much more sophisticated, since it is completely transparent to the programmer. To implement pointer swizzling correctly, every thread keeps an internal pointer to each of the minipages through the desired view. On each page fault, instead of swizzling the program pointer itself, we swizzle the internal pointer of the issuing thread that corresponds to the faulted minipage. However, the next time the same program pointer is used, it holds the old value and not the new one. Therefore, each pointer dereferencing in the program is always performed through the thread's internal pointer. In this way, the correct view is always used. This is enabled with the help of the instrumentation described in Section 5.

4. VARIABLE-SIZE DETECTION UNIT

As was mentioned before, the detection unit in our implementation is dynamic. More precisely, we detect races in granularity of minipages and not in granularity of a fixed number of bytes. For this purpose, each minipage is associated with the information essential for the data race detection algorithms—the access history for Djit$^+$and the current candidate set and diagram state for Lockset.

The race detection mechanisms are activated in the page fault handler, at the same place where the pointers to minipages are swizzled. The handler is supplied with all the necessary information. From the fault type, the access type is deduced, and from the faulted address, the corresponding minipage and respective view are calculated. Thereafter, the access history and the lockset state of the minipage are retrieved and the detection mechanisms invoked.

A single minipage can contain primitive types consisting of bytes or words, as well as more complex user types. As will become clear in Section 5 (describing instrumentation),

our detection granularity is at least the size of an entire object defined by the classes and structures of the program. In fact, in all modern object-oriented programming languages the objects tend to be small and self-contained, consisting of only strongly related data fields. Thus, the object granularity is indeed the proper granularity to be employed.

Though our implementation is entirely transparent, we still give a programmer the ability to fine-tune the detection in order to adjust it to his or her specific needs. Thus, while a single minipage may contain a single object, several objects can be aggregated into a larger detection unit to occupy a single minipage. A single minipage may even contain entire arrays. In the case of arrays, it is also possible to locate each of the array elements on a separate minipage.

It should be emphasized here that splitting an array across several minipages still allocates all of its elements in one contiguous area, exactly as is done by the original C++ allocation routines. The division to minipages is only logical and it allows pointing to different elements of the array through different views. In this way, it controls the size of the unit in which data race detection is performed. Clearly, detecting races for each element separately imposes greater overhead than testing for the races in bunches. If the entire array is placed on single minipage, it will resemble a large object, with views swizzled for all the elements at once. Obviously, this also minimizes the additional space that is needed for the data race detection algorithms. It can, however, become a source for false alarms, when different threads access non-overlapping regions of the array.

Nevertheless, the granularity of detection has quite a useful and important property—a race free program at some given granularity will not introduce any races at any finer granularity. Thus, it is a good idea to first locate all elements of some array on one minipage; only if alarms are reported, should one try splitting it into several units. If alarms still appear, the detection granularity can be further refined until either all alarms are determined false, or the data race is discovered. Note that refining the detection granularity in this way is a programmer-directed process, which also involves high overhead. Therefore, it should be activated only in debugging mode, when the programmer suspects certain alarms to indicate real races.

In order to allow splitting array over several minipages, the `malloc` and `operator new[]` functions are supplied with additional parameters. The first is just the number of requested elements (not used in `malloc`). The second parameter controls the number of subsequent array elements to be placed on each minipage. If this parameter equals 1, then each element resides on a separate minipage. If it equals the number of requested elements, then all elements are placed on one minipage. Every intermediate value between these two limits is also acceptable. Since, obviously, this value is user defined, it is specified by the programmer through the use of code annotations. An example of the use of an overloaded `operator new[]` function is depicted in Figure 6.

Original code:

```
Type* arr1 = new Type[5];
Type* arr2 = new Type[7];
Type* arr3 = new Type[num];
```

Instrumented code:

```
⇓ The whole array is on one minipage
Type* arr1 = new(5, 5) Type[5];
⇓ Each element is on a separate minipage
Type* arr2 = new(7, 1) Type[7];
⇓ The array is on two minipages
Type* arr3 = new(num, num/2) Type[num];
```

Figure 6: Example of the use of an overloaded operator new[] function to split arrays over several minipages

5. INSTRUMENTATION HIGHLIGHTS

In this section we describe how the user's program should be instrumented so that the race detection algorithms can be correctly activated. For our instrumentation to work properly, we require that the entire program code be available and compile correctly prior to being changed. Under these conditions, we show how the instrumentation task can be completed transparently by an automatic preprocessing phase. We also guarantee that after our modifications are completed, the program will still compile and run correctly.

For brevity's sake, in what follows we give only the main ideas of the instrumentation. The exact details appear in the full version of the paper.

Every class or structure `Type` in the source code of the program is forced to inherit from our `SmartProxy<Type>` template class. This class has only public functions, henceforth called *smart functions*, and no data members. Clearly, such class hierarchy only expands the functionality of class `Type`. The basic idea of our instrumentation is that the smart functions, when applied on a shared object or a pointer to it, locate the corresponding minipage and return the reference or the pointer to the object through the correct view. These functions are called, respectively, `smartReference()` and `smartPointer()`. The required minipage is calculated from the `this` pointer passed as an implicit argument to these smart functions. Then, the thread's internal pointer (Subsection 3.3) is used to return the correct view. Note that the pointer or the reference returned by a smart function always refers to exactly the same object on which the function was originally invoked. Thus, further access to its data members or functions is still possible.

To enable the primitive types (`int`, `double`, etc.) to support the smart functions described above, we created wrapper classes. During the instrumentation process we substitute each potentially shared appearance of a primitive type in the source code with a corresponding fully functional wrapping class, which has the full set of smart functions.

The instrumentation of global and static objects and arrays is supported as well. During the program initialization phase, after these objects have already been constructed in the data segment, they are copied to our shared space. All accesses to the objects are then redirected to occur only through our copies.

Finally, in the case that the source code is not available, we simulate the reading and/or writing of those minipages which we suspect will be affected by the uninstrumented code. For this purpose, the `SmartProxy<Type>` class contains two additional smart functions—`read` and `write`. When invoked on an array, for example, these functions "touch" all elements from the starting address of the array to its very last element (unless the maximum number of elements to be touched is specified).

An example of an instrumented function is depicted in Figure 7.

Original code:

```
void func( Type* ptr, Type& ref, int num ) {
   for ( int i = 0; i < num; i++ ) {
     ptr->data +=
                    ref.data;
     ptr++;
   }
   Type* ptr2 = new Type[20];
   memset( ptr2, 0, 20*sizeof(Type) );
   ptr = &ref;
   ptr2[0] = *ptr;
   ptr->member_func( );
```

Instrumented code:

```
void func( Type* ptr, Type& ref, int num ) {
   for ( int i = 0; i < num; i++ ) {
     ptr->smartPointer()->data +=
                  ref.smartReference().data;
     ptr++; ⇐ No access to shared memory
   }
   Type* ptr2 = new(20,2) Type[20]; ⇐ 2 elements/minipage
   memset( ptr2->write(20), 0, 20*sizeof(Type) );
   ptr = &ref; ⇐ No access to shared memory
   ptr2[0].smartReference() = *ptr->smartPointer();
   ptr->member_func( ); ⇐ The code of the invoked
                          member function is instrumented
```

Figure 7: Example of an instrumented function

6. REPORTING RACE ATTRIBUTES

6.1 Obtaining Race Attributes

In the MultiRace system each reported data race and each announced locking discipline violation are identified by the following attributes:

1. The memory address of the raced object;
2. The minipage that contains the object (the address above points inside this minipage);
3. The instruction completing the data race or the locking discipline violation;
4. In the case of a data race, a previously logged instruction with which the current instruction is concurrent.

In order to be able to report the exact pairs of racing instructions, the access history of every minipage (see Subsection 2.3.3) holds, in addition to the time frames, the *instruction pointers* (a.k.a. IPs) of the most recently logged read and write in each of the threads. (The IP is available from the information supplied by the operating system to the page fault handler.) Hence, when a data race is detected, the addresses of both racing instructions are at hand.

In the full paper we present an additional technique in which the exact pairs of source file and line at which the possible races occur can be determined during runtime. In this way, later examination of the source code will reveal the contents of the problematic instructions.

6.2 Reporting Races

We offer two methods for reporting races. In the first, a report is made each time a data race is detected. This is done by invoking a software *breakpoint interrupt*, `int 3`, which freezes all threads and starts a debugger. The programmer can then query the faulted address and look in the annotated source code (available from program's debug database) for the locations of the conflicting instructions. The programmer can also retrieve additional information, such as a thread's stack contents and the values of relevant global variables. After the source of the race has been identified, the programmer can resume the execution of the program and locate further races.

In the second method, all the relevant information is saved for each data race that occurs during the program's execution. During the system's finalization phase, the debugger is invoked and the list of all the detected races is also reported. The programmer can then traverse this list of detected races and learn their causes directly from the source code.

In contrast to data races, we cannot be sure whether the locking discipline violations are real bugs. Therefore, rather than suspending the program whenever such violations occur, we save all their relevant information. When the debugger is invoked, either when a data race is detected or during the finalization phase, this list can be traversed and all violations easily located.

7. OPTIMIZATIONS

We analyzed several benchmark applications and, as expected, the two main sources of overhead were the smart functions and the page faults. In this section we suggest several optimizations that reduce both the number of faults and the number of smart functions invoked during program execution. Additional optimizations are presented in the full version of the paper.

7.1 Loop Optimizations

Figure 8 shows two very efficient optimizations applied to a simple loop. OPT1 can be used only if both arrays reside entirely on corresponding single minipages and no synchronization appears inside the loop. If these restrictions are not applied, real races can be missed. In this optimization, we distinguish between the first access and all successive accesses. Thus, we economize on the invocation of smart functions and on the need to locate corresponding minipages and views each time the elements of arrays are accessed.

The OPT2 optimization further extends the idea of OPT1 to arrays that occupy several minipages. The only restriction in this case is that the elements must be accessed sequentially, without any synchronization inside the loop. The OPT2 optimization first simulates write and read accesses to each of the minipages occupied by both arrays and only then executes the original code. This optimization becomes most efficient when the number of elements in each array is large and the number of minipages these elements occupy is relatively small. The reason is that only the corresponding minipages are traversed, rather than all the elements of the array. In addition, there is no need to locate all the minipages occupied by the array. Only the first minipage need be located and the rest processed sequentially. Hence, OPT2 shows good speedups even when the number of elements per minipage is relatively low.

7.2 Changing the Granularity

Figure 9 demonstrates the reduction in slowdowns for a race-free version of the FFT application, when the granularity of source and destination matrices is changed from 1 complex number per minipage to 256 (the overheads were calculated relatively to the original uninstrumented version with the corresponding number of threads). The figure also depicts the overheads for a situation in which each matrix is

ORG — Original code:

```
for ( i = 0; i < size; i++ )
   arr1[i] += arr2[i];
```

BAS — Code after initial instrumentation:

```
for ( i = 0; i < size; i++ )
   arr1[i].smartReference() += arr2[i].smartReference();
```

OPT1 — Code after distinguishing the first access from successive accesses. Accessing the first elements of each array through smart functions and the rest through regular pointers:

```
if ( size > 0 )
   arr1[0].smartReference() += arr2[0].smartReference();
for ( i = 1; i < size; i++ )
   arr1[i] += arr2[0];
```

OPT2 — Same code as original, except for first simulating write and read accesses to all elements of corresponding arrays. The size of the array is passed to the `write()` and `read()` functions, so that only the correct number of minipages is affected:

```
arr1->write( size );
arr2->read( size );
for ( i = 0; i < size; i++ )
    arr1[i] += arr2[i];
```

Figure 8: Loop optimizations

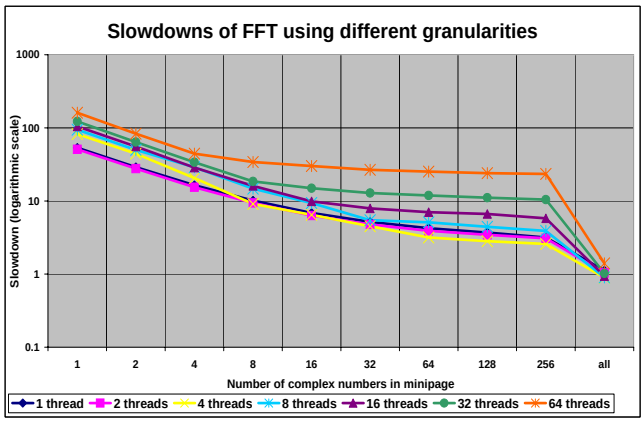

Figure 9: Overheads of a race-free FFT benchmark application with different granularities

entirely located on a corresponding single minipage. In this case, the overheads drop sharply.

8. MEASURED OVERHEADS

In this section we present the MultiRace overheads measured for six classical benchmark applications: Integer-Sort (IS), Water-nsquad (WATER), LU-contiguous (LU), Fast Fourier Transform (FFT), Successive Over-Relaxation (SOR) and the Traveling Salesman Problem (TSP). For evaluation of overheads we used the data-race-free versions of the applications. Therefore, we were able to place each of the allocated arrays on single minipages (the default configuration of MultiRace). Table 1 shows some characteristics of these applications.

We performed our measurements on the Microsoft Windows NT operating system, running on a 4-way IBM Netfinity server (550MHz) and 2GB of RAM. We tested the application using 1, 2, 4, 8, 16, 32 and 64 threads. The original non-instrumented versions behaved nicely, meaning that the best execution times were achieved with four threads and

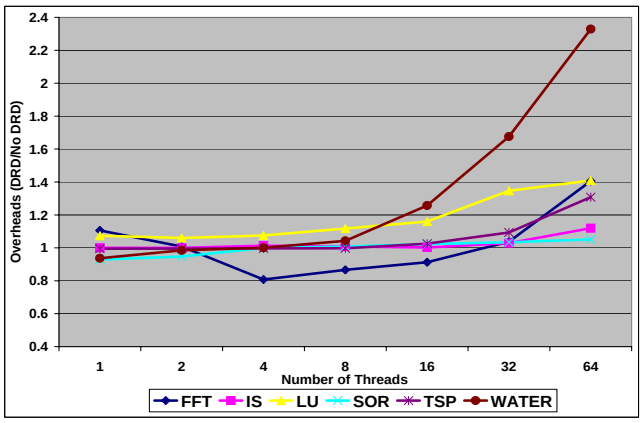

Figure 10: Overheads with/without data race detection (DRD) measured for the 6 benchmark applications

took about 25% of the execution time with only one thread. This suggests that the applications were programmed correctly and that they are highly suitable for this kind of benchmark. We were glad to find that our instrumented versions containing the data race detection mechanisms exhibited the same speedups, indicating that we did not introduce too much noise into the applications.

In fact, for some applications, especially for a low number of threads, the instrumented versions with all the data race detection mechanisms run even faster than the original unaltered applications. In most cases, the reason for this rather unexpected behavior stems from the differences between the standard allocation routine and our version. Our allocation method was found to be more suitable for a multithreaded environment than the one implemented by the standard `malloc` and `operator new` functions. The reason is the access pattern of the applications to large arrays, which causes lots of misses in the caches of the executing processors. This is explained in more detail in the full version of the paper.

Figure 10 presents the overheads obtained for our benchmark applications (without taking into account the time required to initialize the system). The overheads seem to be low and steady for 1–8 threads. The applications have either very low overheads or none at all. This suggests that our system is scalable in the number of CPUs. However, most of the applications suffer from heavier overheads for a higher number of threads. The reason is that the access logging and race detection mechanisms have to be activated separately for each of the running threads. In addition, more threads require more inter-thread communication. All these result in more time frames, more page faults, and thus more work performed by MultiRace per benchmark execution.

Figure 11 shows the breakdowns of the overheads for the tested benchmark applications. The breakdowns are presented according to the overhead imposed by the addition of instrumentation and memory request interception, the supplementary overhead caused by the write and read page faults required to record the accesses, and the overhead from adding the data race detection algorithms. From these breakdowns it can be seen that most of the overhead is caused by the page faults, while the overheads due to the

	Input Set	Shared Memory	Number of Minipages	Write/Read Faults	Number of Time Frames	Time in sec (no DR)
FFT	$2^8 * 2^8$ matrix	3 MB	4	9/10	20	0.054
IS	2^{23} numbers, 2^{15} values, 15 repeats	128 KB	3	60/90	98	10.68
LU	$1024 * 1024$ matrix, block size $32 * 32$	8 MB	5	127/186	138	2.72
SOR	$1024 * 2048$ matrices, 50 iterations	8 MB	2	202/200	206	3.24
TSP	19 cities, recursion level 12	1 MB	9	2792/3826	678	13.28
WATER	512 molecules, 15 steps	500 KB	3	15438/15720	15636	9.55

Table 1: Different characteristics of the benchmark applications (for two threads)

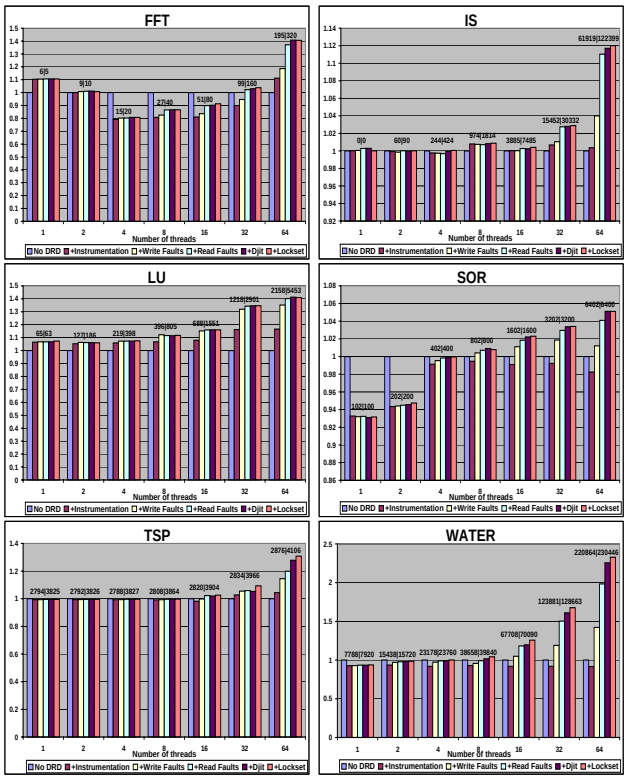

Figure 11: Breakdowns of the overheads measured for the benchmark applications. The times are relative to the original uninstrumented version with the corresponding number of threads. The numbers above the bars indicate, respectively, the number of write and read faults.

data race detection algorithms are much lower.

In addition to the breakdowns, Figure 11 shows the number of write and read faults for the corresponding number of threads[2]. It is easy to see that for all the benchmark applications except TSP, the number of faults increases linearly with the number of threads. This increase also exists in TSP, but it is less evident because of the different path cut-offs TSP performs. These cut-offs increase TSP's dependency on thread scheduling and make its behavior less deterministic than that of other applications.

The linear growth in the number of page faults causes the work performed by Djit$^+$ to increase quadratically with the number of threads. This is because each access in the faulting thread is always checked against the most recent accesses in each of the other threads. The Lockset algorithm, however, exhibits only a linear growth.

9. RELATED WORK

Netzer and Miller formally defined and characterized the different types of data races in [15]. They also showed how to further improve the data race detection accuracy [14]. They introduced a two-phased method that helps to distinguish feasible data races from artifacts that only appear as a result of earlier races. In the first phase apparent data races are detected. In the second, postmortem phase, they are validated. In this way, either each detected race is guaranteed to be feasible, or, when insufficient information is available, sets of races are identified within which at least one is guaranteed to be feasible.

In [5], Dinning and Schonberg empirically compared two algorithms for data race detection in parallel Fortran programs—English-Hebrew labeling and Task Recycling. These methods also realize the Lamport's happens-before relation, but they are different from the vector-time [12] that we used in our work, and are more suitable for the fork-join paradigm. The overheads of the Task Recycling algorithm, which was found to be more efficient, ranged from 150% to over 1000%.

Hood et al. presented in [8] a combined approach for PCF Fortran programs that coordinates static analysis with an on-the-fly data race detection algorithm. The static information collected during program analysis was used for minimizing the number of checks during the execution, as well as for reducing the number of false alarms. To further improve performance, they limited their detection to programs without nested parallelism and optimized the algorithm at the cost of completeness and accuracy of race detection. By this, they managed to get only a 40% overhead.

Savage's Eraser, described in [17], uses binary rewriting and is intended for lock-based synchronization only. Instead of detecting data races according to the happens-before relation, it uses the *lockset refinement* to find violations of locking discipline. Since the technique is too conservative (not every violation of the discipline is necessarily a race), Eraser suffers severely from the false alarm problem. In addition, it uses a fixed-size unit for detection—a fact that

[2]We apologize for the small font size in this figure, imposed by space limitations. The interested reader can learn the numbers from the full paper.

further reduces its accuracy. The tool typically slows down applications 10 to 30 times.

Ronsse and Bosschere implemented a non-intrusive on-the-fly data race detection tool, which is a combination of execution record/replay with an on-the-fly detector [16]. They suggested that races should be located using two "equivalent" executions. In the first phase, a trace of the order of all synchronization operations is created, which is then used in the second phase to replay the execution and detect data races (in fixed granularity). In their implementation, the first recording phase imposed only a minor overhead of 2% on average. The second replay phase, however, slowed down typical applications 30 times on average (up to 80 in the worst case).

10. CONCLUSIONS

Until recently, on-the-fly data race detection in multithreaded environments was considered to be very inefficient and highly imprecise. Hence, in all currently available techniques there is a tradeoff—a reduction in runtime overhead and space requirements results in an increase in the number of missed races and/or the amount of false detection. To the best of our knowledge, all dynamic data race detection tools are restricted to detection in fixed size granularity. This further decreases their accuracy.

In this paper we suggest a framework called MultiRace—an efficient transparent tool that combines two very powerful algorithms for on-the-fly data race detection at object-size granularity. By employing optimized and extended versions of Djit$^+$ and Lockset, MultiRace detects respectively a subset and a superset of all the raced shared locations in the execution of a program. For many types of programs this ensures that most of the possibly raced locations are detected, while guaranteeing that all data races that actually occurred will be reported. Because of this attribute, MultiRace is unparalleled by any other available on-the-fly detection technique.

In order to detect races in granularity of program objects, MultiRace takes advantage of a transparent source code instrumentation approach. This allows to perform compile-time static analysis, to employ different optimizations and, most importantly, to use fine-grain detection on global and static objects as well as on dynamically allocated data. In addition, MultiRace makes novel use of memory mappings and pointer swizzling. This further simplifies the access logging and the data race detection mechanisms.

Data race detection makes it easier for the programmer to trust the program. It also spares the necessity of adding synchronization "just in case". By logging a small portion of all the accesses, MultiRace imposes only a minor overhead on the tested program. Using MultiRace, accurate, efficient, and transparent data race detection can be performed while the program is executing in production mode.

11. REFERENCES

[1] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. Technical report, University of Wisconsin, Sept. 1992.

[2] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA'91)*, pages 234–243, May 1991.

[3] V. Balasundaram and K. Kennedy. Compile-time detection of race conditions in a parallel program. In *Proceedings of the 3rd International Conference on Supercomputing*, pages 175–185, June 1989.

[4] T. Brecht and H. Sandhu. The Region Trap Library: Handling traps on application-defined regions of memory. In *USENIX Annual Technical Conference, Monterey, CA*, June 1999.

[5] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, Mar. 1990.

[6] P. Emrath and D. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 89–99, May 1988.

[7] C. Flanagan and S. Freund. Detecting race conditions in large programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001)*, pages 90–96, June 2001.

[8] R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Proceedings of the 1990 Conference on Supercomputing*, Nov. 1990.

[9] A. Itzkovitz and A. Schuster. Multiview and Millipage—fine-grain sharing in page-based DSMs. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, pages 215–228, Feb. 1999.

[10] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordechai. Towards integration of data race detection in DSM systems. *Journal of Parallel and Distributed Computing (JPDC), 59(2)*, pages 180–203, Nov. 1999.

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

[12] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms, Elsevier Science Publishers, Amsterdam*, pages 215–226, 1989.

[13] R. H. B. Netzer and B. P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *1990 International Conference on Parallel Processing, 2*, pages 93–97, Jan. 1990.

[14] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *Proceedings of the 1991 Conference on the Principles and Practice of Parallel Programming*, pages 133–144, Apr. 1991.

[15] R. H. B. Netzer and B. P. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems, 1*, pages 74–88, Mar. 1992.

[16] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems, 17(2)*, pages 133–152, 1999.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems, 15(4)*, pages 391–411, Oct. 1997.