

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

Eli Pozniansky Assaf Schuster
Computer Science Department
Technion—Israel Institute of Technology
`{kollega,assaf}@cs.technion.ac.il`

Abstract

Data race detection is 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 work 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`, as well as its implementation details. The paper also proposes some alternatives to and optimizations of `MULTIRACE`. It shows that the overheads imposed by `MULTIRACE` are often much smaller (orders of magnitude) than those obtained by other existing dynamic techniques.

KEYWORDS: Data Race, Multithreading, Concurrency, Synchronization, Instrumentation.

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) [28]. These are the real races that might happen in any specific execution of a program. According to [26], 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* [28]. These are 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 in [25] 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 (in fact, it is undecidable in the presence of loops or recursion).

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 [5, 10, 12, 22] 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 defined by the structures and classes of the program. However, the main drawback of these techniques 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 [3, 25, 27] 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, the on-the-fly methods [7, 8, 9, 16, 31, 34, 35, 36, 38] 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 detected races.

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, different input is supplied, other data races might pop up. Hence, in order to detect all data races, all possible execution paths should be considered. This, however, is not practical in most cases. Therefore, the dynamic check should not be restricted only to program testing time, but 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 becomes the cause of a 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 co-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 objects¹. We remark that the average object size is known to be very small (about 20 bytes for Java objects), hence the detection accuracy on objects should be 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 work 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 [16, 38], called DJIT⁺. The algorithm is based on Lamport's *happened-before* partial order relation [18], and it is capable of efficiently detecting apparent races as they occur. The second technique is an improved variant of the LOCKSET algorithm [35], 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.

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

The benefit of combining these two 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 apparent 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* [6, 15], 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 in places where accesses to shared locations are performed. It should be emphasized here that in contrast to many currently available data race detection techniques, which require some kind of help from the programmer, the detection in MULTIRACE is completely transparent. It does not require any assistance, except for some specific adjustments and configurations needed to fine-tune the detection.

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 [7, 8, 30, 31, 34, 35, 36], 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 presents the assumptions used in our system model, and gives a definition of data races. Section 3 presents the DJIT⁺ detection algorithm and provides a proof of its correctness. Section 4 shows the extended LOCKSET algorithm and describes how it integrates with DJIT⁺ in same framework. Section 6 discusses all the implementation issues of the logging mechanism in MULTIRACE: it describes the memory organization and the management of views. Section 7 explains our notion of the variable-size detection granularity. Section 8 gives the highlights of the instrumentation that enables the access logging. Section 10 describes how MULTIRACE is integrated within the tested program so that data races are correctly reported. Section 11 suggests several optimizations of MULTIRACE, and Section 12 presents the obtained overheads. Section 13 surveys related works. We give our conclusions in Section 14.

2 System Model and Definitions

2.1 Memory and Synchronization Models

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

The most common and easy to understand model for shared memory is *sequential consistency* [19]. 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, limits the employment of many optimizations that would be possible otherwise. Thus, modern memory models weaken the restrictions of the memory behavior.

In this work one of our challenges was to reduce the number of restrictions and requirements from the memory model. For this purpose, we henceforth assume that our system obeys some *weakly ordered* memory model. An example, to such a model is the *data-race-free-1* shared memory model definition, first presented in [2]. 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, and the executing threads can disagree on the values they read.

In order to prevent concurrent accesses and avoid data races, the threads use *synchronization primitives*. The employment of some synchronization operation on a shared *synchronization object* S is called a *synchronization event*. We follow the basic observation that each synchronization object involves some *logical token*. The token is *released* by one set of threads that reach a certain point in their execution and is *acquired* by another set of threads. Once all the members of the corresponding releasing set have released the token, members of the acquiring set are allowed to proceed. We thus discuss synchronization events in terms of *release* and *acquire* operations between corresponding sets of threads.

For example, an `unlock(S)` operation releases a token of S and a corresponding `lock(S)` operation acquires it; a global `barrier(S)` operation causes all threads to release the logical token of S as they reach the barrier, and only when the token has been released, causes them to re-acquire the token. 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. It is only necessary to correctly define when the tokens are acquired and released, and what are the threads involved in the corresponding synchronization operations.

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 [18]. This definition also resembles those presented in [3, 16, 30]; by expressing the causality relation between program events, it allows recognition of apparent data races that pop up in some specific

program execution. By this definition the races can be detected and announced when they actually take place in the program under examination. However, different executions may result in different sets of races, and the absence of detected races in the current execution does not imply that the whole program is data race free.

Definition 1 *The happens-before partial order, denoted \xrightarrow{hb} , is defined for all computational events (reads, writes) and synchronization events (releases, acquires) that happen in a specific execution, as follows [18]:*

Program Order: *If α and β are any two events performed by the same thread, with α preceding β in the program order, then $\alpha \xrightarrow{hb} \beta$. (The program order is imposed by the program source code.)*

Release and Acquire: *If α is a release and β is its corresponding acquire, both operating on the same synchronization object S , then $\alpha \xrightarrow{hb} \beta$.*

Transitivity: *If $\alpha \xrightarrow{hb} \beta$ and $\beta \xrightarrow{hb} \gamma$, then $\alpha \xrightarrow{hb} \gamma$.*

The motivation for this definition should be clear—we want to be able to distinguish between those pairs of accesses that are potentially concurrent and those that are synchronized. Therefore, we further define:

Definition 2 *We say that two events α and β are synchronized if either $\alpha \xrightarrow{hb} \beta$ or $\beta \xrightarrow{hb} \alpha$. 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, α and β , to the same shared location, 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.*

Figure 1 shows an example in which two threads synchronize by means of a lock.

3 DJIT⁺

In order to detect data races, we present an algorithm called DJIT⁺, which is a revised version of the earlier DJIT algorithm [16, 38]. 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. To make things as clear and as correct as possible, we give all the DJIT⁺ proofs from scratch, without the restricting assumptions.

DJIT⁺ relies on a formal framework called *vector time frames*, which is based on Mattern’s virtual time vector time-stamps [20]. 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.

In the next subsection we describe how vector time frames can be used to realize the \xrightarrow{hb} relation. In later subsections, we present the full algorithm and the proof of its correctness.

Thread 1	Thread 2
LOCK L	
WRITE X,1	
UNLOCK L	
WRITE Y,2	
	LOCK L
	WRITE Y,3
	UNLOCK L
	WRITE X,4

Figure 1: The *happens-before* partial order: In the depicted execution, the UNLOCK L by Thread 1 is a *release* which corresponds to the LOCK L by Thread 2, which is an *acquire*. Hence, according to Definition 1, the UNLOCK by Thread 1 \xrightarrow{hb} the LOCK by Thread 2. By program order and transitivity, the WRITE X,1 by Thread 1 \xrightarrow{hb} the WRITE X,4 by Thread 2, and these operations are thus synchronized. In contrast, the WRITE Y operations form a data race, since according to the same definition, they are concurrent.

3.1 Realizing the \xrightarrow{hb} Relation

3.1.1 Time Frame Vectors

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 an upper bound on the number of threads is known in advance, and stored in a constant called *maxThreads*. Each thread *t* maintains a vector of time frames, denoted $st_t[.]$, having *maxThreads* entries (*st* is an abbreviation of “synchronization times”). For the sake of simplicity we assume that the IDs of the threads are positive integers in the range of $[0, \text{maxThreads}-1]$. For each index *u*, the entry $st_t[u]$ stores the latest local time frame of thread *u*, whose release operation is “known” by thread *t* to have happened before *t*’s latest acquire. Clearly, the entry $st_t[t]$ is the current local time frame of *t*, and it is equivalent to the number of releases actually made by *t*.

During the execution, the vectors of the threads are maintained in the following way:

- If *t* performs a release, it increments its own time frame:

$$st_t[t] \leftarrow st_t[t] + 1$$

- If *u* acquires the token 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:

```
for  $i=0$  to  $\text{maxThreads}-1$ 
 $st_u[i] \leftarrow \max( st_u[i], st_t[i] )$ 
```

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 . Thus, each synchronization object S also maintains a similar vector. The actual propagation of values during a *release-acquire* chain is done in three steps. Upon a release of synchronization object S by t , the thread increments its own time frame. Then, each entry i in the vector of S , denoted $st_S[i]$, is updated to hold whichever value is greater: its own or that of $st_t[i]$. When another thread u acquires S later, it updates each $st_u[i]$ entry to hold the greater value: its own or that of $st_S[i]$.

In this way, u not only modifies its vector to hold the current time frame of t , but also modifies its “knowledge” of the local time frames of other threads according to the information stored in t 's vector. This knowledge is propagated through a chain of corresponding *release-acquire* pairs.

Next, we show that the \xrightarrow{hb} relation can be verified from the vectors of the time frames we defined above:

Claim 1 *Let α be an access at thread t_α performed during time frame T_α . Let β be an access at thread t_β ($\neq t_\alpha$). Then $\alpha \xrightarrow{hb} \beta$ iff at the moment when β occurs, $T_\alpha < st_{t_\beta}[t_\alpha]$.*

Proof If $\alpha \xrightarrow{hb} \beta$, and since α and β are in distinct threads, then there exists a chain of releases and corresponding acquires, from the first release in t_α to the last acquire in t_β , so that $\alpha \xrightarrow{hb} \text{first release}$, $\text{first release} \xrightarrow{hb} \text{last acquire}$, and $\text{last acquire} \xrightarrow{hb} \beta$. According to the vector updates we defined above, the information about t_α 's local time frame, $st_{t_\alpha}[t_\alpha]$, propagates through this chain, reaches t_β , and is stored in $st_{t_\beta}[t_\alpha]$. Thus, it follows that $T_\alpha < T_{\text{first release}} \leq st_{t_\beta}[t_\alpha]$.

If $T_\alpha < st_{t_\beta}[t_\alpha]$, then there exists a sequence of corresponding *release-acquire* instruction pairs, which transfer the local time frame's value from t_α to t_β , finally resulting in t_β “hearing” that t_α entered a time frame, which is later than T_α (otherwise $st_{t_\beta}[t_\alpha]$ could not be updated to its current value). This same sequence can also be used to transitively apply the \xrightarrow{hb} relation from α to β . \square

Suppose that our logging mechanism (discussed in the next subsection) is capable of logging all accesses in all threads, without missing any. Also recall that we are interested in locating data races as they occur in the program execution. 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 the DJIT⁺ algorithm.

3.1.2 Requirements from the Logging Mechanism

In order to correctly detect data races on weak memory systems there are several restrictions on the logging mechanism that we use in our proofs. First, our logging mechanism should agree with the partial order imposed by the \xrightarrow{hb} relation. In other words, for every two accesses, α and β , if $\alpha \xrightarrow{hb} \beta$ then α is also logged prior to β . Otherwise, some accesses that should appear inside a critical section might be logged outside it, or we could find a corresponding release and acquire instruction pair, such that the acquire precedes the release in the log. This would obviously contradict our definition of the \xrightarrow{hb} relation.

Thread 1	Thread 2
⋮	⋮
RELEASE(L1)	
WRITE X,1	
RELEASE(L2)	
⋮	RELEASE(L3)
	WRITE X,2
	ACQUIRE(L2)
	WRITE X,3
	⋮

Figure 2: There is a data race between WRITE X,1 in Thread 1 and WRITE X,2 in Thread 2. If the log is not coherent, then after WRITE X,1 was already logged by Thread 1 and at the moment WRITE X,2 is logged in Thread 2, the history of X, seen by Thread 2, is not yet updated. Thus, the race is undetected.

Second, we suppose that for every shared location the threads perform the action of both logging an access and testing it for races with previously logged accesses (to the same location) in a mutually exclusive way (or, in other words, *atomically*). Thus, for every two accesses to the same location v , α and β , it holds that either α is logged prior to β , or β is logged prior to α . Together with the previous restriction this leads to the following observation: if α and β both access the same shared location and if α is logged prior to β , then $\beta \not\stackrel{hb}{\rightarrow} \alpha$. This is the reason that in our proofs we are able to distinguish the first access in a time frame to each of the shared locations (Claim 2), and we can tell which of the time frames occurred first (Claim 3).

Third, we remark that in order to detect races as they occur, the memory behavior of the log should be *coherent*. In other words, we assume an existence of a separate agreed-among-all-threads order on all the accesses to the history of each shared location. Under this order, the reads of the history always return the most recently written value to it. Otherwise, if we suppose that the memory behavior of the log is weak, then at a data race occurrence each thread might see a different history. Hence, some of the data races can remain undetected. Figure 2 demonstrates an example of this observation.

Note that using a logging mechanism that satisfies the above requirements will not necessarily impose the coherency or the sequential consistency assumptions, which we promised to avoid in Subsection 2.1 (recall that we want to employ a weak memory model). For example, in order to achieve the coherency of the log, the history of each shared variable can be protected from concurrent reads and updates by a special *log-oriented* synchronization implemented in hardware, or by some “light-weight” synchronization primitive specially designed for this purpose. Furthermore, as will become obvious from the proofs, we are completely indifferent to the actual values that are written to or read from the shared locations of the tested program. We also do not impose any restrictions on the order in which updates made by one thread to

some shared location are seen by the other threads. In fact, the actual order can differ from the one implied by our log. (However, both of them should agree with the \xrightarrow{hb} partial order, and be sequentially consistent in total absence of data races.) Hence, we do not care about the existence of an agreed-among-all-threads order, as required by the sequential consistency or the coherence models. Therefore, the assumption of a weakly ordered system still remains. This is an additional advantage of DJIT⁺ over the original DJIT algorithm, and it also stands for our improved version of LOCKSET discussed in Section 4.

3.1.3 Data Race Detection Predicate

Next, we obtain the main theorem, which implies the correctness of the data race detection protocol:

Theorem 1 *Let α and β be two accesses to the same shared location in respective threads t_α and t_β ($t_\alpha \neq t_\beta$) during respective local time frames T_α and T_β . Assume that α was logged and tested for races prior to the occurrence of β (i.e., α appears before β in our log). Then α and β are concurrent iff at the time when β is logged, it holds that $T_\alpha \geq st_{t_\beta}[t_\alpha]$.*

Proof If $T_\alpha \geq st_{t_\beta}[t_\alpha]$, then, by Claim 1, $\alpha \xrightarrow{hb} \beta$ does not hold. Since α was logged beforehand and β is just currently being logged, then (according to previous subsection) it cannot hold that $\beta \xrightarrow{hb} \alpha$. Therefore, by Definition 2, α and β are concurrent.

In the other direction, if α and β are concurrent, then $\alpha \xrightarrow{hb} \beta$ does not hold, and thus, again by Claim 1, $T_\alpha \geq st_{t_\beta}[t_\alpha]$. □

Note that if, in addition, at least one of the accesses α or β defined above is for writing, then α and β , according to Definition 3, form a data race.

We assume that the logging mechanism can record for each access its type (read or write), its issuing thread's ID, and its issuing thread's local time frame at the moment of the access. In addition, we assume the existence of an up-to-date vector of time frames per thread. The algorithmic aspect of Theorem 1 is encapsulated in the following predicate, \mathcal{P} . \mathcal{P} receives two accesses, α and β , such that:

1. Each access is defined by a triple (*access_type*, *thread_id*, *time_frame*), as specified above (henceforth abbreviated by a triple (*type*, *tid*, *tf*)).
2. α and β are issued in different threads and they access the same shared location v .
3. α was already logged and tested for races with previously logged accesses, while β has just been logged.

\mathcal{P} evaluates to ‘true’ iff α and β form a data race. Formally,

$$\mathcal{P}(\alpha, \beta) \triangleq (\alpha.type = \text{write} \vee \beta.type = \text{write}) \wedge (\alpha.tf \geq st_{\beta.tid}[\alpha.tid]).$$

3.2 Reducing the Number of Checks

We have thus far assumed that all accesses are recorded by the logging mechanism. In addition, we have supposed that it is necessary to check each newly logged access with all previously logged accesses. An algorithm that does so, even if correct, will probably impose high overhead on the system.

We are interested in detecting data races and pinpointing the exact locations of the instructions involved. But more importantly, we are interested in determining whether some given execution exhibits races or is data race free. 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 the algorithm's correctness.

3.2.1 Which Accesses to Check?

We make the following observation:

Claim 2 *Consider an access α in thread t_α during time frame T_α , and accesses β and γ in thread $t_\beta=t_\gamma$ ($\neq t_\alpha$) during time frame $T_\beta = T_\gamma$. Assume that γ precedes β in the program order. Then, if α and β are concurrent, α and γ are concurrent as well.*

Proof Let f_β and f_γ denote the respective values of $st_{t_\beta}[t_\alpha]$ when β and γ occur. Since $st_{t_\beta}[t_\alpha]$ is monotonically increasing, and γ precedes β in the program order, we know that $f_\beta \geq f_\gamma$. Since $\alpha \xrightarrow{hb} \beta$ is false, we know by Claim 1 that $T_\alpha \geq f_\beta$. Thus, $T_\alpha \geq f_\gamma$, and by Claim 1 we get that $\alpha \xrightarrow{hb} \gamma$ is false.

Let f_α denote the value of $st_{t_\alpha}[t_\beta]$ when α occurs. Since $\beta \xrightarrow{hb} \alpha$ is false, we know by Claim 1 that $T_\beta \geq f_\alpha$. Since $T_\gamma = T_\beta$, we get that $T_\gamma \geq f_\alpha$, and so, by Claim 1, $\gamma \xrightarrow{hb} \alpha$ is false.

Since we showed that neither $\alpha \xrightarrow{hb} \gamma$ nor $\gamma \xrightarrow{hb} \alpha$ hold, we know that α and γ are not synchronized, or, in other words, they are concurrent. \square

The above claim implies 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. In fact, these first accesses act as the “representatives” of the corresponding time frames. Similarly, it is sufficient to apply the predicate \mathcal{P} only to pairs of accesses to the same location which are the first in their respective time frames.

More precisely, suppose that in some execution we log in thread t_α an access α_v to some shared location v . Also suppose that α_v is the first access in its time frame, T_α , to v . Then, if α_v is a write, we may avoid further logging of accesses to v during the remainder of T_α . If α_v is a read, then further logging of reads of v during the remainder of T_α is redundant; however, it is still necessary to log the first write to v in T_α . After the first (if any) write is logged, there is no need to log additional accesses to v until the next time frame in t_α begins.

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.

We have now ascertained that only the first write, and the first read if it comes prior to a write, to each shared location in each time frame should be logged and checked. Thus, in our further discussion we will consider these first accesses only.

3.2.2 Which Time Frames to Check?

We make an additional observation:

Claim 3 *Assume that in thread t_α an access α is currently being logged. Assume also that in thread $t_\beta = t_\gamma$ ($\neq t_\alpha$), prior to the occurrence of access α , our logging mechanism recorded an access β in time frame T_β and another access γ in time frame T_γ , so that $T_\beta < T_\gamma$. Then, if α is concurrent with β , it is also concurrent with γ .*

Proof Access α is concurrent with β ; hence, by Theorem 1 we get $T_\beta \geq st_{t_\alpha}[t_\beta]$. Since $T_\gamma > T_\beta$ and $st_{t_\alpha}[t_\beta] = st_{t_\alpha}[t_\gamma]$, we get that $T_\gamma > st_{t_\alpha}[t_\gamma]$. Therefore, again by Theorem 1, we get that α is concurrent with γ . \square

In fact, Claim 3 shows that if currently logged access α in thread t_α forms a data race on a shared location v with some previously logged access β in thread t_β (as defined by the claim), then it certainly forms a data race with all later accesses in t_β to the same location v . An equivalent and more important claim is that if there is no data race between α and γ (as defined by Claim 3), then there can be no data races between α and accesses that appear prior to γ in the program order. Therefore, if α and γ are synchronized, the pair α - β need not be checked.

Developing the observation above, 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 .

We pay readers attention to the fact that detecting races in the described manner has an additional advantage, besides the obvious performance gain. Since we detect races only between the pairs of most recent accesses in different threads, we do not overwhelm the programmer with an enormous number of races, of which only one may be of real importance. For example, suppose that until some point in an execution, thread t_β performed k accesses to some shared location v — $\beta_1, \beta_2, \dots, \beta_k$. Then another thread, t_α , performed an access α to the same location v , and this access completed a data race with β_1 . In such a case, according to Claim 3, the access α races with each of the accesses β_i . Reporting all α - β_i pairs is redundant—the α - β_k pair alone supplies sufficient information to deduce that v is raced, and that a possible reason for this is missing synchronization between α and β_k .

Remark: Note that if accesses α and γ (as defined by Claim 3) form a data race, then α might also form a data race with accesses that precede γ in the program order (as happens in the above example

with the $\alpha\text{-}\beta_k$ pair). It would be impossible to find these data races before α is actually performed and logged, which is the very point when they become apparent. By our definitions, although these accesses precede γ , their races with α occur simultaneously to the race of α and γ . Therefore, we choose to announce only the $\alpha\text{-}\gamma$ pair, which is the closest pair of accesses that form a data race. However, the algorithm can be easily refined to detect and report races with earlier accesses as well. This will clearly require maintaining a deeper history of accesses and will probably impose a greater overhead on the system.

3.3 The Detection Protocol

In order to implement the detection protocol, each thread t and each synchronization object S hold a vector of time frames, as described in Subsection 3.1. 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, and the threads always read a consistent state of the values that were recently written.

We assume that we are indeed able to detect the first accesses to each of the shared locations in each time frame. In Section 6 we describe a framework that actually enables us to do so. Figure 3 shows the full DJIT⁺ protocol. It detects races as they occur between the first accesses to each shared location in each time frame. In addition, it minimizes the number of pairs of time frames that need be checked.

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. This is one of the advantages of DJIT⁺ over original DJIT, which was capable of detecting only the very first race in an execution.

Remark: Note that the above algorithm remains correct even if the threads do not perform the action of both logging and testing in a mutually exclusive way. The reason is that the “concurrent-with” relation is symmetric (i.e., if α is concurrent with β , then also β is concurrent with α). In addition, according to the algorithm, the issuing thread first updates its entry in the access history of v , and only then reads the entries related to other threads (and all these accesses are atomic and coherent). Thus, if there exists a pair of racing accesses, α and β , to some shared location v , for which the access history is also concurrently accessed (i.e., before both logging and testing are completed in one thread, the other thread accesses the access history), then it is guaranteed that either α will discover β , or β will discover α as participating in a race. In the worst case, both accesses will be first logged and then tested for races, such that both $\alpha\text{-}\beta$ and $\beta\text{-}\alpha$ pairs will be detected. In such a case, the latter pair can be easily eliminated from the data race report.

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 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 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 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 $ar_v[u] \geq st_t[u]$. In other words, t checks all reads as well as all writes by other threads to v .

Figure 3: The full DJIT⁺ protocol

4 LOCKSET

In this section we propose a refined and optimized version of the LOCKSET algorithm, first presented in [35]. Our implementation takes 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 the barriers synchronization primitive, so that it can be integrated with DJIT⁺ within the same framework.

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 4: The refined LOCKSET algorithm

The improved LOCKSET algorithm was actually implemented in the MULTIRACE system (fully described in Sections 6–12), where it exhibited excellent outcomes and added only negligible overheads to the tested programs.

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 [35], 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 is depicted in Figure 4.

This process is called *lockset refinement*. It ensures that any lock that consistently protected v so far is contained in $C(v)$. Clearly, a lock l is in $C(v)$ if until this point in the execution, every thread that accessed v was holding l at the moment of access. If, in addition, l has consistently protected v during the entire program execution, it will remain in $C(v)$ until the program terminates.

Note that there is a distinction between reads and writes in the depicted algorithm that does not exist in the original LOCKSET described in [35]. 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

...	...
UNLOCK L	UNLOCK L
...	...
LOCK L ₁	LOCK L ₁
LOCK L ₂	WRITE X,1
WRITE X,1	LOCK L ₂
WRITE X,2	WRITE X,2
UNLOCK L ₂	UNLOCK L ₂
UNLOCK L ₁	UNLOCK L ₁
...	...

(a) The sets of locks held during both writes to X are identical. (b) X is consistently protected only by lock L₁, which is already held during the first write to X.

Figure 5: These examples show that it is possible to omit the refinement of candidate sets during the second write to X.

concurrently. However, the first write to v permanently removes *readers_lock* from $C(v)$. Thus, for $C(v)$ not to become empty, another “real lock” is required to consistently protect v .

Remark: Note that similarly to DJIT⁺, the LOCKSET algorithm can operate correctly on weak memory systems. The necessary requirement is that for every shared location v the candidate set $C(v)$ will be coherent.

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⁺. Figure 5 demonstrates this idea.

The original LOCKSET takes into account only locks, which can be *locked* or *unlocked*. It recognizes neither the “low-level” *release* and *acquire* operations that we used in DJIT⁺, nor the concept of time frames. We take advantage of a fact that each *lock* and each *unlock* correspond respectively to exactly one *acquire* and one *release*. In addition, we observe that the reaching of a *barrier* by all threads involves a *release* with a later *acquire* by each thread. Thus, in order to be able to reason about the execution of the LOCKSET algorithm in terms similar to those of DJIT⁺, we define that a new time frame starts with each invocation of an *unlock* or a *barrier* operations.

Claim 4 Consider two accesses, α and β , to some shared location v , such that (1) they are both in the same thread t , (2) α precedes β in the program order, and (3) $T_\alpha = T_\beta$ (i.e., the accesses occur during the same time frame). Then $Locks_\alpha(v) \subseteq Locks_\beta(v)$, where $Locks_\theta(v)$ is the set of locks acquired by t during access θ to v .

Proof Since $T_\alpha = T_\beta$, there are no *unlock* or *barrier* operations between α and β , yet there can appear any number of *lock* operations. Each *lock* expands the set of locks held by a thread, and only *unlock* reduces it. Since α precedes β in the program order, it follows that $Locks_\alpha(v) \subseteq Locks_\beta(v)$ \square

The direct results from Claim 4 are:

1. For any set of locks L , it holds that: $L \cap Locks_\alpha(v) = L \cap Locks_\alpha(v) \cap Locks_\beta(v)$.
2. For any set of locks L , it holds that: $L \cap Locks_\alpha(v) \subseteq L \cap Locks_\beta(v)$.
3. From (2) it follows that if $L \cap Locks_\beta(v) = \emptyset$, then $L \cap Locks_\alpha(v) = \emptyset$.

Now consider some general algorithm that detects possible races through intersections of sets of locks that are held by different threads during accesses to shared locations. In other words, the algorithm checks whether on access θ to some shared location v , $L \cap Locks_\theta(v) \stackrel{?}{=} \emptyset$, where L is any set of locks. According to results 1–3 of the claim above, such an algorithm will not obtain any additional information by checking accesses that are not first in their respective time frames.

The LOCKSET algorithm falls almost entirely into the category of such algorithms. There is only one pitfall. Recall that in our version of LOCKSET, when $C(v)$ is updated, a read access adds a special *readers_lock* lock to $locks_held(t)$. Thus, a read followed by a write in same time frame seems to contradict Claim 4 above (whereas a write followed by a read obviously does not). This problem does not exist if both accesses are logged and tested for possible races, as is done in DJIT⁺. Nonetheless, it should already be clear that by recording only these first accesses, a significant portion of all the checks can be omitted and the overhead significantly reduced.

4.3 Supporting Barriers

In Subsection 2.1, we assumed the existence of a system model that involves both two-way and global synchronization primitives—locks and barriers. The original LOCKSET, as well as the refined version discussed so far, do not take advantage of the latter primitive. In this subsection we propose an additional refinement which enables the employment of barriers.

Recall that the definition of a barrier entails a suspension of any thread that reaches the barrier, other than the last one. The last thread to reach the barrier awakens all the other threads and continues its own execution. Hence, there cannot be 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, and a new time

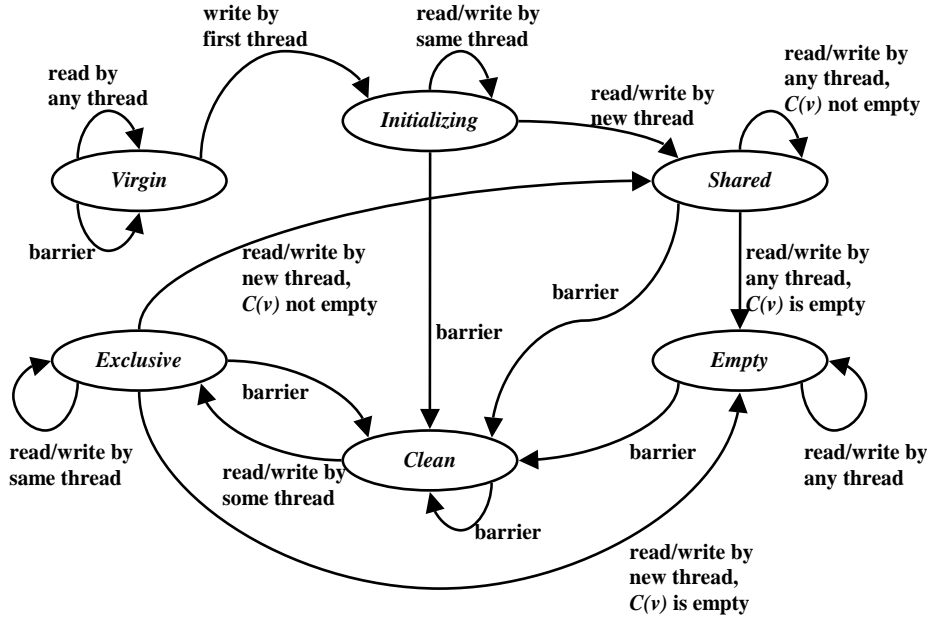


Figure 6: The state transition diagram used in our extended LOCKSET. Each shared location keeps its state, which changes when various threads read and write the location or when a barrier is reached by all threads. $C(v)$ is updated only in *Shared* and *Exclusive* states. Race warnings are reported only the first time *Empty* state is reached from the other states.

frame initiated. Afterwards, the detection should be restarted as if a new execution has just begun. However, the inventors of the original LOCKSET suggested suspending the refinement of the candidate sets for uninitialized variables. This was done in order to support the common programming practice of initializing shared variables without first protecting them with any locks. Since the reaching of a barrier by all threads does not really begin a new execution, suspending the detection in the manner suggested above is not fully correct and may lead to missing some data races. Therefore, a more sophisticated method is required.

To achieve the task, we further expand the ideas presented in the original LOCKSET. In addition to the candidate set, we employ for each shared location a state transition diagram, which acts as some sort of heuristic. This diagram controls the refinement and the maintenance of the candidate sets, as well as the announcement of race warnings. We expand the original diagram of LOCKSET to include the notion of barriers. Figure 6 illustrates our extended state transitions diagram.

When a shared location is first created or allocated, its state is set to *Virgin*, indicating that it has not yet been referenced by any of the threads. The first time the location is written, it enters the *Initializing* state, meaning that it is currently being initialized by exactly one thread. Each subsequent access by the same thread leaves the state unchanged and does not update $C(v)$.

Since the semantics of the program are unknown, it is not so simple to detect when some variable becomes initialized. Hence, we assume that a shared variable is initialized when it is first accessed by a

second thread or when the thread that first wrote to it reaches a barrier. Thus, if the location is accessed by a second thread, it reaches the *Shared* state, in which $C(v)$ is updated for the first time. When and if $C(v)$ becomes empty, the state is changed to *Empty* and a race warning is issued. After the *Empty* state is reached, each later access to same shared location leaves the state unchanged and does not generate a race warning.

If all threads reach a barrier, every shared location that has already been initialized, changes its state to *Clean*, and a new time frame is initiated. The only shared locations that maintain their state are those in *Virgin*. 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 4). When location v is first accessed after the barrier, it reaches the *Exclusive* state. The pair of *Clean-Exclusive* states behaves similarly to the *Virgin-Initializing* pair of states—the only difference is that in the *Exclusive* state, 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), in which case the state again becomes *Empty*. Otherwise, the state changes back to *Shared*, meaning that at least two threads correctly access and modify v .

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 [35]. 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.

5 Benefits of Combining LOCKSET and DJIT⁺ in One Tool

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 (Figure 7).
- 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 (Figure 8). (However, despite this insensitivity, different sets of races might still appear if the program takes another execution path when, for example, different input is supplied.)
- Since every data race is also a violation of the locking discipline, for many types of programs it

F ← 0	
Thread 1	Thread 2
WRITE X,1 LOCK L WRITE F,1 UNLOCK L	LOOP: LOCK L WRITE T,F UNLOCK L IF (READ T = 0) GOTO LOOP; WRITE X,2

Figure 7: Neither thread holds any locks during the accesses to X, thus causing LOCKSET to produce a warning. Note that the WRITE X,2 operation by Thread 2 is always executed only after the flag F (initially set to zero) is raised by Thread 1. Therefore, the writes to X are always ordered and properly synchronized, with a *release-acquire* chain between them. Hence, in this specific execution, LOCKSET’s warning on X can be easily classified as a false alarm using the additional information obtained from DJIT⁺.

Thread 1	Thread 2
∴ WRITE Y,1 LOCK L WRITE X,2 UNLOCK L ∴	∴ LOCK L WRITE X,3 UNLOCK L WRITE Y,4 ∴

Figure 8: There exists a feasible data race between the WRITE Y,1 and WRITE Y,4 accesses. Since in this specific execution order there is a *release-acquire* chain between these accesses, DJIT⁺ cannot detect the race. In contrast, LOCKSET correctly locates a violation of the locking discipline, and reports about it.

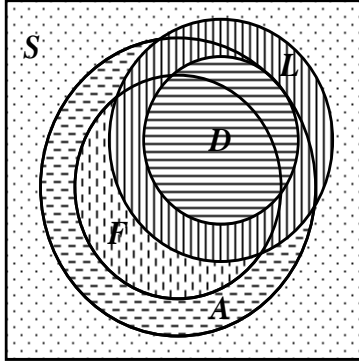


Figure 9: The set of all shared locations in some given program P is represented by region S . The set of shared locations in P that are participating in feasible data races is represented by region F . The set of shared locations in P that are participating in apparent data races is represented by region 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 a subset of both A and L . We remark that for many types of programs (e.g., such that are not *completely nondeterministic* [10]), A usually becomes a subset of L .

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 9 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. (However, it should continue logging the accesses.)
- The warnings generated by LOCKSET can be “ranked” with the help of the *happens-before* relation realized in DJIT⁺. For example, if some access to shared location v empties $C(v)$, but there exists extensive synchronization between this access and a previous one, then this pair most likely will not form a feasible race under different thread scheduling. In this way, every race warning reported by LOCKSET can be given a grade that reflects the probability of being a real race.
- Since both detection algorithms require same access logging mechanism, most of the overhead involved with the logging is paid only once.
- Eraser, in which the original LOCKSET algorithm was first implemented, typically slows down applications by a factor of 10 to 30 [35]. As we show in Section 12, monitoring and checking only the first accesses in time frames drastically improves the performance of LOCKSET.

6 Implementation of the Logging Mechanism

So far we have discussed several algorithms that help the programmer to detect data races and locking discipline violations in executions. These algorithms assumed the ability to log all accesses to each of the shared objects as they occur.

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. Next, we give the idea of how the program code should be instrumented so that the logging mechanism is correctly activated. Finally, in later sections we propose several optimizations and present overheads measured in `MULTIRACE`.

6.1 View 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 [14, 15], [6] and [38], 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 is depicted in Figure 10.

6.2 Swizzling between Views

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 (like IRIX, AIX, LINUX, Solaris, Windows NT 4.0, Windows 2000 and Windows XP) 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 obtained, appropriate action can be taken: the faulted minipage can be located, its view tested and modified, and the race detection mechanisms invoked. Afterwards, the faulting instruction can be re-executed and the execution continued.

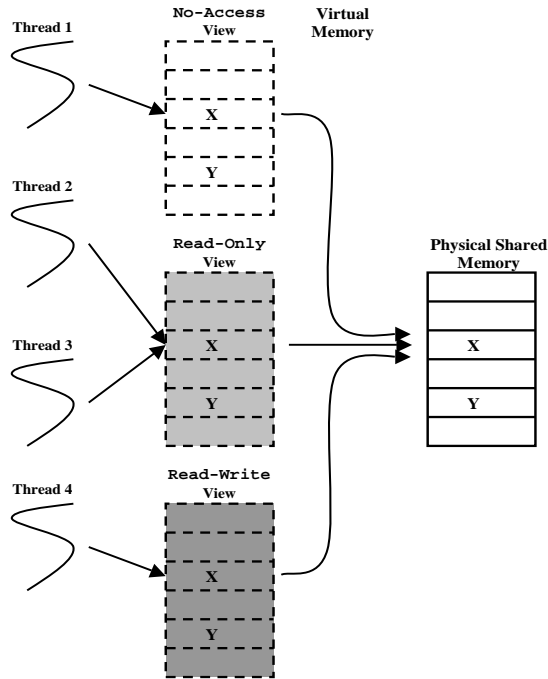


Figure 10: The memory layout and the depiction of views. Note that different threads can access the same shared location X through different views, thus getting different protections for X.

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 [6, 38]. Each thread maintains a pointer for each minipage (see Subsection 6.4). This pointer can refer to the minipage through one of the three views. 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.

What is less obvious (at least at this stage), but very important, is that such a logging mechanism allows many different optimizations to be employed. As will become clear from the sections describing instrumentation (Section 8) and optimizations (Section 11) issues, it is possible to significantly reduce the amount of the detection code injected into the tested program and, therefore, to decrease the overhead involved.

Remark: Note that when a page fault occurs, the register used to reference the faulted memory address is already loaded with the wrong value. Therefore, when the fault handler returns, the faulted machine instruction is re-executed and the wrong address is reused. Hence, in addition to swizzling the pointer and the view, the referencing register must be swizzled as well. In order to know exactly which register to swizzle, the faulting machine instruction is disassembled. Then the referencing register is updated to hold the correct value of memory address through the new correct view.

6.3 Memory Layout and Memory Allocations

The memory layout is depicted in Figure 10. In order to achieve such a layout, 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 memory requests of the program, 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.

Recall that shared locations are considered to be all global and static objects, as well as all dynamic allocations from the heap. In what follows, we do not distinguish between these three types and treat all of them as if they were dynamic allocations. The reason for this will become clear later, when in Section 8 we show how the program is instrumented to achieve this goal.

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 allocation request creates a minipage; its starting address, returned to the programmer, can reside in each of the three views. In fact, the address returned from the allocation request is always through the `ReadWrite` view, so that constructors, for example, are invoked without being faulted.

Allocation of memory and creation of minipages are sequential, according to the order of invocations. Hence, the i ’th minipage in the shared memory area corresponds to the i ’th object allocated. Moreover, the offset of the i ’th minipage in each view is equal to its offset from the start of the shared memory area (Figure 11).

6.4 Minipage Representation

In order to maintain information about minipages, all threads share a global array (Figure 11). Each entry in it corresponds to one minipage and stores shared information related to it. This information includes the size of the minipage and its offset from the start of the shared memory area in each of the views (recall that all offsets are equal). The entries in the array are ordered by the minipage offsets from the start of the view. This property is used later, when minipages are searched by their offset.

Since we detect races in units of minipages, each entry holds, in addition, all relevant information for the detection algorithms—the access history for `DJIT+`, and the current candidate set and state for `LOCKSET`. The entry also keeps some extra information, to be discussed later.

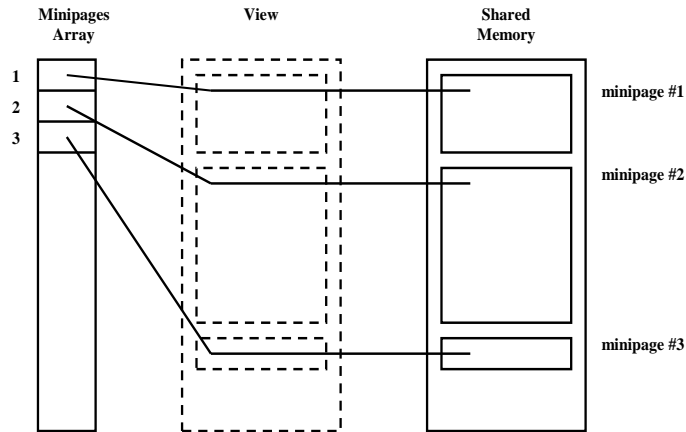


Figure 11: Memory allocation and minipage creation are sequential, according to the order of requests.

To implement the pointer swizzling technique described in Subsection 6.2, every thread has to keep a pointer to each of the minipages through the desired view. Therefore, in our implementation, each thread maintains a private array, in which each entry corresponds to one minipage and points to the beginning of that minipage through the correct view.

The global array and all the private arrays share the same index for each minipage. In other words, the i 'th entry in each array corresponds to the i 'th minipage that was created (Figure 12). This simplifies the implementation and releases the system from the need to search or compute the indexes. In addition, each index uniquely defines a minipage. This means that every minipage is recognized by its index and knowing the index provides all the information associated with that minipage. It is necessary only to read the entries from the global and private arrays that correspond to it.

Each time some thread declares or allocates a shared object, enough space to contain the object is reserved in the shared memory area. If the i 'th minipage is allocated, an entry with index i is created in the shared array of minipages. In addition, an i 'th entry is created in each of the private arrays of the threads. These entries consist of a pointer to the start of the corresponding minipage through the `NoAccess` view.

6.5 Pointers to Different Offsets in a Minipage

In the discussion so far, we did not impose any restrictions on the size of a minipage or on the number of objects contained in it. In fact, as already mentioned, entire arrays can be allocated on a single minipage. Since we want to allow referencing all entries in an array and all possible fields of objects, it is necessary to point to different offsets inside the minipage. It is easy to see that the address of each object and subobject located in the shared memory area can be calculated as an offset to the minipage containing the object, plus an additional offset from the start of the minipage (Figure 13).

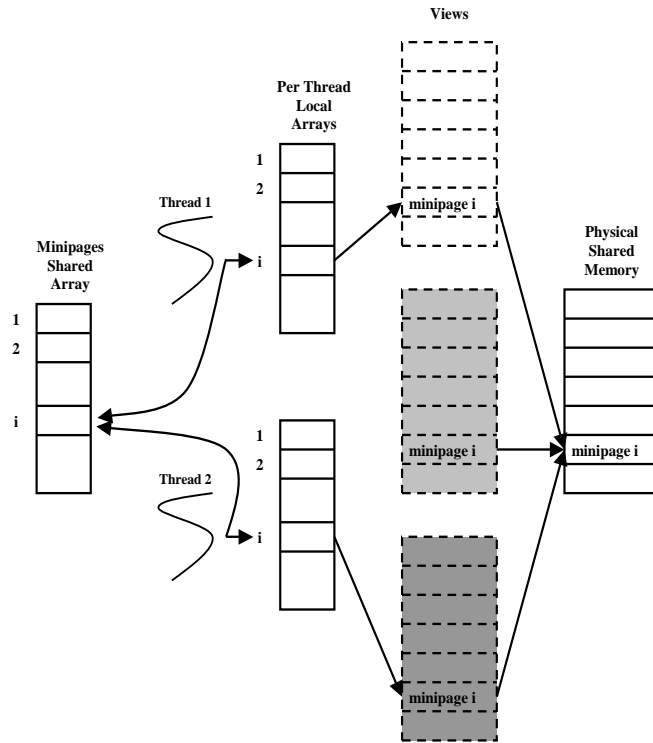


Figure 12: The index i uniquely defines the minipage, view, and region allocated in shared memory area.

6.6 Locating an Object's Minipage and Offset

Programming languages like C and C++ allow pointers to objects, i.e., they allow the memory address at which the object resides to be obtained. Such a pointer, for example, can be obtained with the help of the *address of* operator, represented in these languages by the '&' symbol. In C++ it can also be retrieved from the 'this' pointer passed as an implicit argument to member functions of structures and classes. This read-only pointer always contains the address of the object for which the function was invoked.

Another way to obtain the address of an object or some offset inside it is through the information supplied to the page fault handler. For instance, suppose that there is some object v , which is viewed through the `NoAccess` view. Accessing one of the fields of v will clearly generate a page fault. The handler

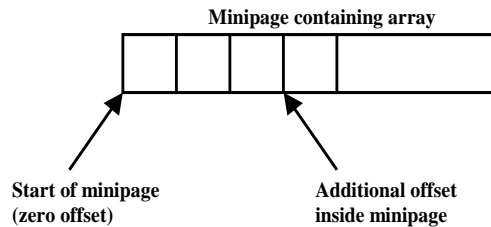


Figure 13: Different offsets inside the same minipage

will get from the operating system the faulting instruction and the faulted address, which will point through a `NoAccess` view to the accessed field of v .

Now suppose that we have somehow obtained a pointer, denoted ptr , which points to some shared object or to an offset inside it. Clearly, this pointer holds an address through one of the three views inside the corresponding minipage, which contains that object. We are interested in obtaining the index of that minipage, so that our data race detection mechanisms can be invoked.

First, we determine to which of the three views the pointer points. This is easily done by comparing ptr with the boundaries of the views, which, of course, do not overlap. Next, we calculate the offset of ptr from the beginning of the corresponding view. Since the memory is allocated sequentially, the entries in the global array of minipages are contiguous. Hence, the global array is binary-searched for an entry M , such that $M.offset \leq offset < M.offset + M.size$, where the index of M identifies the desired minipage. Clearly, this search requires $O(\log N)$ operations, where N is the current number of minipages in the system.

This search can be improved by maintaining a lookup table that holds, for each page of shared memory (4 KB), the index of the first minipage allocated on it. In this way, only the entries corresponding to the page need to be searched. Calculating the page number takes $O(1)$ (the offset is divided by the page size), and the binary search takes $O(\log n)$, where n is the number of minipages residing on that page.

The search can be further improved by maintaining a lookup array that maps each address in the shared memory to the index of the minipage corresponding to it. This makes it even easier to obtain a minipage from the address. It is necessary only to read in $O(1)$ the entry from the mapping array that corresponds to the offset calculated from ptr . Clearly, the problem with this method is the amount of additional memory it consumes. If we suppose that the index of each minipage is represented by a double-word (4 bytes), then the size of the map becomes four times bigger than the size of the shared memory area. Yet for applications with small shared memory requirements this method proves to be the fastest of the three.

The last thing that we may need to compute is the additional offset inside the located minipage M . It is easily calculated by subtracting the starting address of the minipage, denoted $M.offset$, from the offset we have: $additional_offset \leftarrow offset - M.offset$.

6.7 Calculating an Effective Address

Assume that some thread t somehow obtained a minipage's index and it wants to access the object on that minipage. As we described in Subsection 6.5, the index alone is not enough. An additional offset inside the minipage is required. Assume that the thread knows this offset as well.

Let us denote the private array of thread t by $pointers_t[.]$, the index of minipage by $minipage$, and the additional offset inside that minipage by $additional_offset$. The effective address of the desired object through its correct view is calculated by the following obvious formula:

$$effective_address \leftarrow pointers_t[minipage] + additional_offset$$

Note that if the object whose effective address we want to calculate is located in the shared area, then the index of the minipage is legal and can be located, as described in previous subsection. But when

the object is local to a thread (e.g., resides on the stack), this index is undefined. In order to enable “transparent” access to local objects, as if they were shared, the first entries (index 0) of the private arrays in each of the threads are reserved and set to zero, i.e., $\forall t : pointers_t[0] \leftarrow 0$. In addition, all local objects are defined to reside on minipage 0 and the *additional_offset*, when calculated, is set to hold the real address of the object (i.e., $minipage \leftarrow 0, additional_offset \leftarrow ptr$). It is easy to see that the formula for calculating the effective address of shared and local objects remains the same.

6.8 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, incremented or decremented 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 pointer is then swizzled so that it points through the `ReadOnly` or the `ReadWrite` view (Subsection 6.2).

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 handler². 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, described in [6], 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. Instead of swizzling the program pointer itself, we swizzle the entry in the private array of the issuing thread that corresponds to the pointed minipage. On each page fault, the accessed minipage is located and the $pointers_t[minipage]$ (defined in Subsection 6.7) is updated to point to the new correct view. However, this does not solve the problem completely—the next time the same program pointer is used, it holds the old value and not the new one. Therefore, each pointer dereferencing is always done through the thread’s private array in two steps. First we calculate from the program pointer the index of the minipage on which the pointed object resides and the additional offset inside that minipage. We then calculate the effective address as described in Subsection 6.7 to obtain the address through the correct view. This is enabled with the help of the instrumentation described in Section 8.

²Note the difference between the “pointed address” and the “address of a pointer”—the location in memory at which the pointer itself resides.

7 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 8 where instrumentation is described, 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, often become a source for false alarms when, for example, 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.

The idea above can be further extended to the entire program under examination. In those portions of the code in which the programmer is sure there are no undetected data races, maximally aggregated granularity should be used (e.g., an entire array should occupy a single minipage). In contrast, in other portions of the code about which the programmer is not sure yet, a finer granularity should be employed.

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 14.

Original code:

```
Type* arr1 = new Type[5];
Type* arr2 = new Type[7];
num = 10;
Type* arr3 = new Type[num];
```

Instrumented code:

```
Type* arr1 = new(5, 5) Type[5];  $\Leftarrow$  The whole array is on one minipage
Type* arr2 = new(7, 1) Type[7];  $\Leftarrow$  Each element is on a separate minipage
num = 10;
Type* arr3 = new(num, num/2) Type[num];  $\Leftarrow$  The array is on two minipages
```

Figure 14: Example of the use of an overloaded `operator new[]` function to split arrays over several minipages

8 Instrumentation Highlights

In this section we describe how the user's program, written in C++, 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 [32].

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 mem-

bers. 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 6.8) 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 15.

9 Operating System and Compiler

Our implementation of `MULTIRACE` was written in C++ for operating systems based on NT technology (Microsoft Windows NT 4.0 or later) and was compiled with the Microsoft C++ Compiler 6.0 (or later). This combination of operating system and compiler made programming in a multithreaded environment a very easy task. For example, the per thread private arrays were defined using the thread local storage (TLS) mechanism. This mechanism gives each thread its own name space and makes it possible for the same variable name to refer to different memory addresses in different threads. In addition, the structured exception handling (SEH) mechanism made it possible to capture page faults and to supply an appropriate handler for race detection and view swizzling.

The swizzling technique was implemented entirely in user mode (except for the initialization phase). The instrumentation task was carried out using only a simple preprocessing phase, without making any modifications to the functionality of the standard compiler. Instead, we exploited some common attributes of our C++ compiler.

One of the compiler features we extensively used is the *inlining*. The `inline` keyword, appearing before a definition of a function, instructs the compiler to try to replace all function calls with the code of the

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 15: Example of an instrumented function

function body. The `__forceinline` keyword compels it to do so. Such inline expansion usually alleviates the function-call overhead at the potential cost of larger code size. Since our instrumentation technique might involve a function call on almost every shared access, the overhead was significantly reduced when inlining was in use.

Finally, we believe that by integrating the ideas presented in this work within the standard compiler and debugger, we can significantly reduce the complexity of instrumentation and hence the total overhead. In this way, the explicit use of smart proxies, smart functions, and smart pointers will become unnecessary. Instead, all the instrumentation will be done during the binary generation. In addition, many new and more powerful optimizations will become possible by employing deeper code analysis. One example of possible optimization is all the thread local objects that should not be instrumented or monitored. Integration with a debugger should also yield some benefits. By utilizing the debug symbol table, the programmer will obtain more precise information about the detected races (e.g., the names of the raced objects). It will also allow to debug unaltered original code and to detect races at the same time.

10 Using MULTIRACE

10.1 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 index of 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.

Since data race detection algorithms are activated only when page faults occur, the memory address of the raced object is easily obtained from the additional information supplied by the operating system to the page fault handler. The corresponding minipage is also easily calculated from this same address, as described in Subsection 6.6.

Obtaining the instruction that completes the possible race is not much harder. The page fault handler in which the detection is performed is supplied with the full set of machine register values from the moment of the fault. One of them, usually called an *instruction pointer*, or IP, holds the exact memory address of the instruction that generated the fault (on the 32-bit Intel architecture on which we performed our tests, this register is called *extended IP*, or EIP).

Getting the exact pair of racing instructions is slightly more complicated. In order to be able to report such pairs, the access history of every minipage (see Subsection 3.3) holds, in addition to the time frames, the IPs of the most recently logged read and write accesses in each of the threads. Hence, when a data race is detected, the addresses of both racing instructions are at hand.

10.2 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 (described in previous subsection) is saved for each data race that occurs during the program's execution. During the system's finalization phase (after all global data has been freed), the debugger is invoked. Using the *watch* feature of debugger, which allows

viewing values of variables that appear in the current scope, the programmer can then traverse the list of all detected data races and learn their causes from the annotated 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 and invoking the debugger 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.

10.3 Suspending and Disabling Detection

Recall that our algorithms are based on the explicit synchronization, while the actual semantics of the program are not known. Therefore, we detect only apparent data races, which are not necessarily crucial bugs. For example, an apparent data race can be a real race that does not affect the result of the execution and was introduced intentionally for a performance gain. An apparent race might also be an unfeasible race that appears due to unrefined granularity or due to some privately implemented synchronization.

To reduce the number of such false alarms, we allow both *suspending* and *disabling* the race detection. To use these features, the programmer should annotate those sections of the code in which the races are known to be intentional. After the program is instrumented, compiled and executed, these annotations turn into function calls that suspend and/or disable the data race detection.

It may happen that some access completes an intentional false data race, but a later access in the same thread and the same time frame forms a real data race. Since MULTIRACE checks only the first accesses in each time frame, DJIT⁺ can miss the later race. Therefore, when the detection is *suspended*, the views are not swizzled and every accessed minipage is rechecked. Then, if a data race is detected, it is re-announced as a low-severity warning. Our approach here differs from the common one. Rather than completely disabling the data race detection, we “intensify” it while diminishing its importance. This obviously increases the number of page faults and hence the total overhead. Most importantly, however, it reduces the number of high-severity false alarms and still ensures that no real data races are missed.

The programmer can, of course, completely *disable* the detection if there is no doubt that some of the reported race warnings are irrelevant or false. When the detection is disabled, the views are not invalidated after release operations, so accesses are not logged and checks for races are not performed. All the other work is performed as usual. The only benefit the programmer gets from disabling the detection is performance. However, such a situation can also lead to missing some real races.

MULTIRACE maintains several types of flags, turned on or off through the programmer’s annotations. These flags control the extent to which the detection can be disabled or suspended:

1. A global flag that controls the detection for all minipages in all threads;
2. A per thread local flag that controls the detection for all minipages within the corresponding thread;
3. A per minipage thread-local flag that controls the detection only for the corresponding minipage within the respective thread.

Suitable annotations in different situations make the detection of data races much more flexible. For example, suspending the detection for all minipages in all threads makes it possible to detect races between accesses that are not first in their respective time frames. Thus, for `MULTIRACE` to work correctly and efficiently, we suppose that the programmer understands how to use all these code annotations to fine-tune the detection.

11 Optimizations

We analyzed several benchmark applications and, as expected, the two main sources of overhead were the smart functions (finding the corresponding minipage and then calculating the effective address) 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 a program's execution.

11.1 Loop Optimizations

Figure 16 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.

11.2 Changing the Granularity

Figure 17 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. Since we run these tests on a 4-processors machine, the best times were obtained for 4 threads (see Section 12 for further details). However, for less than 8 complex numbers per minipage, the application has lower slowdowns when running with 1 or 2 threads than with 4 threads. The figure also depicts the overheads for a situation in which each matrix is entirely located on a corresponding single minipage. In this case, the overheads drop sharply.

ORG — Original code:	<pre>for (i = 0; i < size; i++) arr1[i] += arr2[i];</pre>
BAS — Code after initial instrumentation:	<pre>for (i = 0; i < size; i++) arr1[i].smartReference() += arr2[i].smartReference();</pre>
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:	<pre>if (size > 0) arr1[0].smartReference() += arr2[0].smartReference(); for (i = 1; i < size; i++) arr1[i] += arr2[0];</pre>
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:	<pre>arr1->write(size); arr2->read(size); for (i = 0; i < size; i++) arr1[i] += arr2[i];</pre>

Figure 16: Loop optimizations

11.3 Caching Pointers and Minipages

As the number of minipages in the system grows, so does the time required for the invalidation operation, which traverses all minipages and moves their views to `NoAccess`. Henceforth, we propose several optimizations that reduce its overhead.

The first optimization is quite straightforward—we cache, for each thread, the minipages it accessed between successive synchronization operations. Thus, rather than invalidating all the minipages, we update only those that were cached. The overhead of the “caching” operation is completely insignificant—when some minipage is first accessed in a time frame, our page fault handler saves the index of the faulted minipage. The speedup, however, can be considerable, especially when the number of accessed minipages is low and the synchronization rate is high.

The second optimization is less efficient at high sync rates, yet it reaches the same speedup in the average case. In this optimization, we hold an additional global array, whose structure is equivalent to that of the private arrays of threads (Subsection 6.4). The only difference is that its entries never change after being set. Each entry i in this array corresponds to minipage i and points to the beginning of that minipage through the `NoAccess` view. When some thread synchronizes, the whole cached array is copied at once (using the `memcpy` copying routine) over the thread’s private array. Such an invalidation is obviously much faster than moving all the minipages to the `NoAccess` view one-by-one.

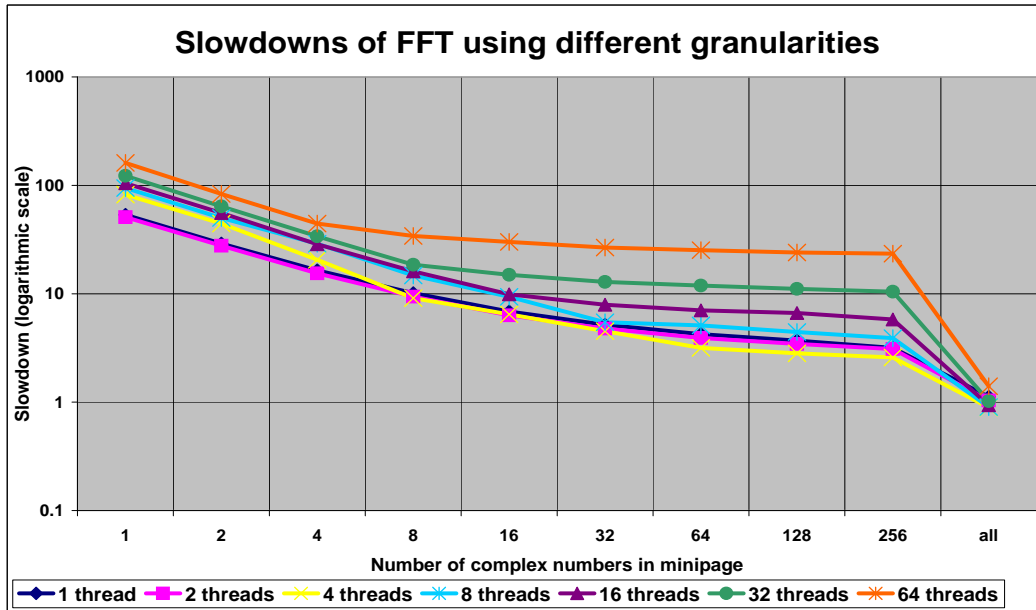


Figure 17: Overheads of a race-free version of the FFT benchmark application, when the granularity (number of complex numbers per minipage) is changed. The overheads are calculated relatively to the original uninstrumented version with the corresponding number of threads.

12 Measured Overheads

In this section we present the MULTIRACE overheads measured for six classical benchmark applications: Integer-Sort (IS) from the NAS parallel benchmarks [4]; Water-nsquad (WATER), LU-contiguous (LU) and Fast Fourier Transform (FFT) from SPLASH-2 [37]; Successive Over-Relaxation (SOR) and the Traveling Salesman Problem (TSP) from Treadmarks [17]. 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 uninstrumented versions behaved nicely, meaning that the best execution times were achieved with four threads and 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.

Figure 18 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 when running with 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

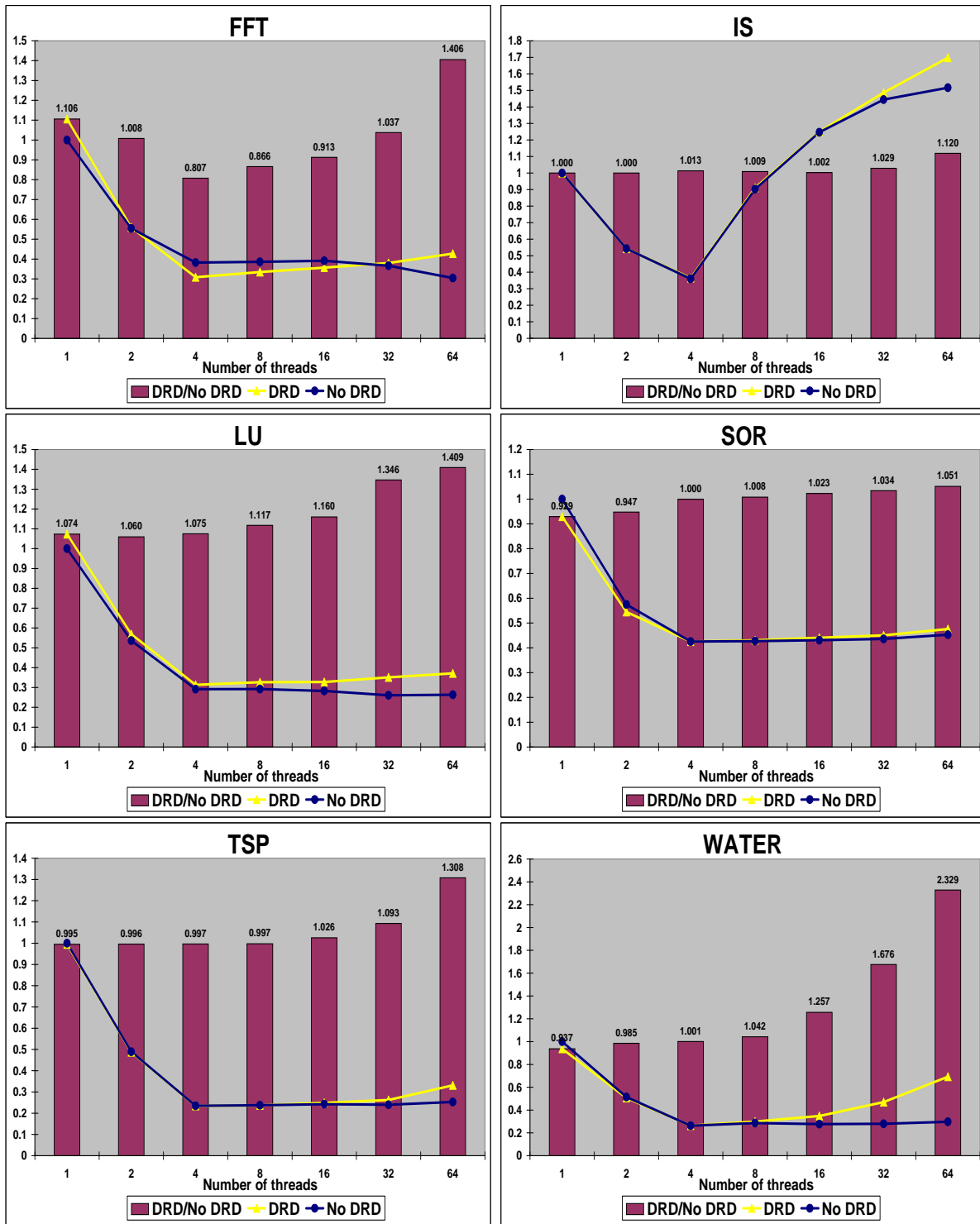


Figure 18: Overheads measured for the 6 benchmark applications using 1, 2, 4, 8, 16, 32 and 64 threads. The times are normalized according to the execution times of the corresponding pure uninstrumented versions with one running thread (i.e., without data race detection). The bars indicate the relative slowdown with/without data race detection (DRD).

	Input Set	Shared Memory Size	Number of Minipages	Number of 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)

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.

When comparing the different overheads, one should consider the following parameters: the shared memory size, the number of shared objects, the granularity unit, the type of synchronization, the number of synchronization points, the number of threads, the number of minipages accessed in each time frame by each thread, and the length of the execution. The shared memory size, the number of shared objects and the granularity unit set the maximal number of minipages. The number of minipages determines, for example, the work required for invalidating the entire memory after a release operation. The number and type of synchronization points set the total number of time frames in threads. The length of the execution, the number of time frames and the number of threads set the average number of time frames in a thread and the average length of a time frame. All these, together with the number of minipages accessed in each time frame, determine the total number of page faults that will occur during the program’s execution.

Figure 19 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 is possible to see that most of the overhead stems from the page faults, while the overheads due to the data race detection algorithms are much lower.

In addition to the breakdowns, Figure 19 shows the number of write and read faults for the corresponding number of threads. 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 (see Subsection 3.2). The LOCKSET algorithm, however, exhibits only a linear growth.

Note that 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. (In fact, the `operator new` is just an invocation of the `malloc` with the requested size.) The reason is the access pattern of the applications to large arrays, which causes lots of misses in the caches of the executing processors.

Any system that uses caches depends on the locality of memory accesses for good performance. Symmetric multiprocessing (SMP) systems that provide separate caches for each processor introduce additional issues that affect application performance. Memory caches must maintain a consistent view of memory for all processors. This is accomplished by dividing the memory into small chunks that make up a *cache line*, and by tracking the state of each chunk present in one of the caches (most current systems have 32-byte cache lines). To update a cache line, a processor must first gain exclusive access to it by invalidating all other copies in other processors' caches. When the processor has exclusive access to the cache line, it may safely update it. If the same cache line is continuously updated from many different processors, that cache line will bounce from one processor's cache to another. Because the processor cannot complete the write instruction until its cache acquires exclusive access to the cache line, it must stall. Such behavior is usually called *cache sloshing*, because the cache line "sloshes" from one processor's cache to another.

One cause for cache sloshing is two or more variables that occupy the same cache line. Updating any of the variables requires exclusive ownership of the cache line. Two processors updating different variables will slosh the cache line as much as if they were updating the same variable. Any function that takes proportionally more time as the number of processors increases is a likely victim of cache sloshing. As more processors compete for the same cache lines, the instructions that access those cache lines run slower and slower. The functions do not actually execute more instructions, but each instruction that needs to wait for the cache takes longer to complete. This increases the total time spent in the function. It is easy to remedy this problem by simply padding data structures to ensure that frequently accessed variables do not share a cache line with anything else. Packing frequently accessed variables together into a single cache line also improves performance by reducing the traffic on the memory bus.

In fact, two variables can share the same line in cache not only if they reside in subsequent memory addresses. Since the size of the cache memory is much smaller than the size of the physical memory, different addresses in the main physical memory can be mapped to the same address in cache. The calculation is preformed by simply taking the lower bits of the physical address that correspond to the size of the cache



Figure 19: 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.

(i.e., modulo cache size). The effects of such a collision can affect a program’s execution time even with one processor and one running thread. For instance, performing subsequent accesses to two variables that are mapped to the same line in cache will always generate at least one *cache miss*. It is estimated that a CPU cache (a.k.a. level one) miss results in a loss of 10-20 clock cycles and an external cache (a.k.a. level two) miss results in a loss of 20-40 clock cycles. As a matter of fact, the description above is not absolutely complete and not fully correct. In most modern systems the collision problem is usually solved by making the cache 2-way or 4-way associative, meaning that every line of memory is mapped to a set of 2 or 4 lines of cache instead of only one. Obviously, this technique reduces the number of collisions, although it does not eliminate all of them and does not entirely remove all the overhead involved.

When we started to test the execution times of the benchmark applications, we found that the pure uninstrumented versions of FFT and SOR suffer from the overheads discussed above. Generally speaking, these applications use two types of arrays—source and destination. In both applications, the arrays are small enough to fit entirely into the physical memory without being swapped to the disk, yet they are big enough to span over several memory pages. The standard `malloc` routine, used to reserve memory for these arrays, aligns the allocations to a boundary of pages (4 KB each). As a result, the modulo cache size of the arrays’ starting addresses are equal, and therefore all their elements in corresponding indexes collide. Thus, even a simple sequential pass over the arrays that reads values from one and writes to the other, generates redundant cache misses and slows down the program. In the original FFT application, according to results presented in Figure 19, both cache sloshing and cache collision seem to combine, thus slowing down the execution. This is especially noticeable for 4, 8 and 16 threads. When running SOR with only one thread, we actually succeeded in eliminating the latter problem by allocating more bytes than necessary for the second array and then moving its start by an offset of 128 bytes. In this way, we ensured that the corresponding indexes of the two arrays were mapped to different lines of cache. This minor change resulted in a speedup of almost 10% for the original version of the application.

In the instrumented versions of the applications, cache collision is less likely to happen. This is because in MULTIRACE the memory is always reserved from an initially allotted memory block, and all allocations are contiguous (Subsection 6.3). For the source and destination arrays to be mapped to the same place in the cache, the size of the first one must be a multiple of the cache size. Their addresses can also “overlap” if there are additional allocations between the arrays, and the total size of these allocations plus the size of the first array is an exact multiple of the cache size. In the tests we performed, this was not the common case.

An additional source of overhead, found when we analyzed the execution times of the pure version of WATER, was the alignment problem. When `operator new[]` is invoked to allocate some array, an additional header of 4 bytes is reserved at the beginning of the memory block (Subsection ??). The original uninstrumented version of the operator’s function calls the standard `malloc` with the total requested size as an argument. Since `malloc` is not aware of the existence of the header, the actual start of the array might thus be located in the middle of a cache line. Moreover, if the allocated array is composed of doubles (or structures of doubles, as in WATER), this 4-byte header might prevent the doubles in the array from being

aligned to their “natural” boundary of 8 bytes. This causes some of them to cross cache line boundaries and hence to generate additional redundant cache faults. When running the uninstrumented version of WATER with only one thread, this situation leads to a performance degradation of about 8%.

In contrast, in MULTIRACE we align each actual start of an array to a 32 byte boundary. If we recognize that a header is needed (see Subsection ??), we make sure to place it to the left of the boundary. In this way, we guarantee that entire cache lines (except for, perhaps, the last) will be filled. We also guarantee that there will be no boundary crossing. Thus, we eliminate the problem described above.

13 Related Work

Works related to on-the-fly detection of data races can be divided into several categories according to a number of criteria. The first is the definition of apparent data races. A common approach is based on Lamport’s happens-before relation [18]. There are also additional approaches, such as the LOCKSET refinement [35].

The next criterion is the kind of synchronization addressed. Some works address the *fork-join* synchronization model [8, 21, 29]. Others focus on lock-based synchronization (two-party synchronization) [7, 35, 36], barrier-only synchronization [33], or both two-party and multi-party synchronization primitives [16, 30]. Post/Wait synchronization is addressed in [13, 24].

An additional criterion is the method used to activate the data race detection mechanisms. Some of the works use the binary or source code instrumentation. In this approach instructions are added either directly to the binary or the source code of the program (prior to compilation), so that during the execution all relevant accesses are logged and detection is invoked. In order to further reduce the cost, works that use instrumentation often employ static analysis of the program [10, 13]. Another approach, usually applied in distributed shared memory (DSM) systems, exploits the consistency protocol, which already provides the proper infrastructure for logging of accesses and data race verifications. Works that depend on the consistency protocol usually do without static analysis of the code and they are more transparent [16, 30, 33].

Another important criterion is the overhead imposed on the tested program. An addition of the detection mechanism, especially in multithreaded systems, can become a bottleneck of the program. In fact, we believe that any on-the-fly detection tool for multithreaded environments will be unscalable in the number of threads when the number of processors is fixed. In contrast, the overheads reported in DSMs usually tend to be much lower, even for the same data race detection logging algorithms. The reason for this is that in distributed systems the overhead imposed by the access logging and data race detection mechanisms usually does not exceed the overhead of the already-existing consistency protocol and communication.

Finally, the detection granularity and accuracy must be considered. Obviously, the intention is to detect all possible data races and reduce the number of false alarms. However, mainly for performance reasons, many currently available tools employ fixed-size detection units. Choosing a small unit results in missing data races, while choosing a large one leads to false detection. Thus, using a variable-size granularity

adjusted to the sizes of the objects in the program is the optimal solution.

Henceforth we discuss in greater detail some of the previous works.

Netzer and Miller formally defined and characterized the different types of data races in [23, 28]. In [25, 27] they showed how to detect data races using a postmortem algorithm and how to further improve the detection accuracy. 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 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 [1, 2], Adve and Hill presented the *data-race-free-1* model that unified four earlier models of shared memory. The only requirement of their model is that the program should appear sequentially consistent in the total absence of data races. However, when the program is not data race free, some of the races can be artifacts that do not occur on a sequentially consistent system, or after-effects of the very first races. The problem of locating races on weakly ordered systems was first discussed by Adve et al. in [3]. For execution on a weak system, they showed how it is possible to either (1) correctly report no data races and conclude the execution to be sequentially consistent, or (2) reason about all detected data races and report only those that appear in a sequentially consistent execution.

In [8, 21, 29], different methods for realizing Lamport’s happens-before relation were described and compared. These methods are different from the vector-time [11, 20] that we used in our work and are more suitable for the fork-join paradigm. The methods introduced in these works maintain, for each thread, a kind of tree of labels, rather than a vector. In [8], Dinning and Schonberg empirically compared two such algorithms for data race detection in parallel Fortran programs—English-Hebrew labeling and Task Recycling. In both cases they used the instrumentation approach in order to install calls to their run-time libraries. 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 [13] 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 logs and 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. In this way, they managed to get only a 40% overhead.

Savage’s Eraser, described in [35], uses binary rewriting and is intended for lock-based synchronization only. Instead of detecting data races according to the happens-before relation, it extends the idea of *lock covers* from [9]. 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 further reduces its accuracy. The tool typically slows down applications 10 to 30 times.

The object race detector for Java programs presented in [36] is closely related to our work. In their

work the inventors employ a refined version of the original LOCKSET algorithm, and use instrumentation (of the JVM byte code) of the Java program to perform data race detection in granularity of objects. In order to reduce the incurred runtime overhead, the inventors use static (escape) analysis of the source code. They also attempt to support the barrier synchronization primitive (mainly for reducing the amount of false alarms reported in the SOR benchmark application), but as a result of that their technique becomes highly dependent on the scheduling order. This, obviously, might lead to missing some of the data races. In addition, as every “only-LOCKSET-based” technique, their method severely suffers from the false alarms problem, and the inability to perform data race detection in higher granularity of portions of array elements (as in our work) complicates this problem. Finally, for similar types of benchmark applications used in [36] and in our work, their tool results in much higher overheads (at least 3 times).

The Brelly algorithm [7], proposed for Cilk programs with locks, detects violations of *umbrella discipline*, which resembles the locking discipline of LOCKSET. Like in Eraser, it suffers from the false alarm problem and typically slows down application 30 to 80 times.

In [6], Tim Brecht and Harjinder Sandhu developed a Region Trap Library (RTL) for trapping of read and write accesses to memory regions in multithreaded environments. This library is quite general and is not designed specifically for the data race detection problem. The RTL approach differs slightly from that of MULTIRACE. Rather than redirecting every pointer dereference through the current view of the minipage, each region in RTL recognizes and maintains all pointers that refer to it. When the protection of the region changes, all pointers referring to it are swizzled to the new value.

Perkovic and Keleher proposed on-the-fly data race detection for lazy release consistency (LRC) distributed shared memory systems [30, 31]. Their method instruments all read and write accesses to shared memory (at word granularity) and checks for data races during the synchronization events. Typically, it causes programs to be up to 6 times slower. Yet this slowdown decreases to less than 2 as the number of concurrent processors grows. Since they use the LRC model and fixed granularity, their method also suffers from the false alarm problem.

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 [34]. They noticed that injecting any detection code into a program might alter its execution schedule enough to make all data races disappear. To overcome this problem, 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). The detection process thus becomes more flexible, allowing more precise exposure of data races and/or step-by-step debugging. 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).

Finally, the Millipage system, a fine granularity page based DSM [15], makes use of a distributed version of the DJIT algorithm to enable on-the-fly detection of the first apparent data race in a program’s execution [16]. The overheads reported for Millipage do not exceed 20% for optimized versions of benchmark

applications. By exploiting the MultiView technique, Millipage prevents false sharing and monitors only those accesses that are relevant for the data race detection algorithm. To allow object-size granularity, Millipage takes advantage of a slightly different approach than the one used in our work—it only intercepts memory allocation routines and creates a minipage (having one of the three protections) for each allocation request. However, it does not support fine-grain detection on global and static objects, and it does not allow transparent splitting of arrays across several minipages.

14 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 work 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.

The commonly used binary instrumentation has its benefits. It does not require the source code, and hence can be used regardless of the language in which the program was written. However, it has the major drawback of not being able to match the detection granularity to the granularity of the variables and objects in the program. Choosing a large detection unit can cause false alarms to appear, while choosing a small one might result in missed races. In addition, if a data race is detected, the source code is still needed in order to locate and correct the problem.

In contrast, to detect races in granularity of program objects, MULTIRACE takes advantage of a much more accurate and transparent source code instrumentation approach. This approach makes it possible 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.

Our implementation is not fully complete. For example, in order to simplify the system, we assume that the maximum number of threads to run is known in advance prior to beginning of the execution. Obviously, when building a full-scale system all such problems should be and can be solved. Another current limitation of our system is that the existing implementation is designed for the C++ language only. However, similar implementations can be developed for other languages as well. The ideal, of course, is to

expand the ideas presented in this work, and to integrate them within all standard compilers, such that the instrumentation, for example, will be carried out together with the machine code generation.

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.

References

- [1] S. V. Adve and M. D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.
- [2] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. Technical report, University of Wisconsin, Sept. 1992.
- [3] 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.
- [4] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmark. Technical report, NASA Ames, Aug. 1991.
- [5] 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.
- [6] 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.
- [7] G. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 298–309, June 1998.
- [8] 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.
- [9] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices, 26(12)*, pages 85–96, Dec. 1991.
- [10] 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.

- [11] C. A. Fidge. Partial order for parallel debugging. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 183–194, 1988.
- [12] 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.
- [13] 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.
- [14] A. Itzkovitz and A. Schuster. Distributed shared memory: Bridging the granularity gap. In *The First ACM Workshop on Software Distributed Shared Memory (WSDSM '99)*, June 1999.
- [15] 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.
- [16] 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.
- [17] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *USENIX Conference*, pages 115–131, Jan. 1994.
- [18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), pages 558–565, July 1978.
- [19] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, pages 690–691, Sept. 1979.
- [20] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms, Elsevier Science Publishers, Amsterdam*, pages 215–226, 1989.
- [21] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputer Debugging Workshop*, pages 1–16, Nov. 1991.
- [22] J. Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 129–139, May 1993.
- [23] R. H. B. Netzer. *Race Condition Detection for Debugging Shared-Memory Parallel Programs*. PhD thesis, University of Wisconsin, 1991.
- [24] R. H. B. Netzer and S. Ghosh. Efficient race condition detection for shared-memory programs with post/wait synchronization. In *International Conference on Parallel Processing, St. Charles, IL*, Aug. 1992.

- [25] R. H. B. Netzer and B. P. Miller. Detecting data races in parallel program executions. Technical report, University of Wisconsin, Aug. 1990.
- [26] 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.
- [27] 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.
- [28] 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.
- [29] I. Nudler and L. Rudolph. Tools for the efficient development of efficient parallel programs. In *First Israeli Conference on Computer System Engineering*, 1988.
- [30] D. Perkovic and P. Keleher. Online data-race detection via coherency guarantees. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 47–57, Oct. 1996.
- [31] D. Perkovic and P. Keleher. A protocol-centric approach to on-the-fly race detection. *IEEE Transactions on Parallel and Distributed Systems (TPDS), 11(10)*, pages 1058–1072, Oct. 2000.
- [32] E. Pozniansky. Efficient on-the-fly data race detection in multithreaded c++ programs. Master's thesis, Technion - Israel Institute of Technology, 2003.
- [33] B. Richards and J. R. Larus. Protocol-based data-race detection. In *Second SIGETRICS Symposium on Parallel and Distributed Tools*, Aug. 1998.
- [34] 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.
- [35] 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.
- [36] C. von Praun and T. R. Gross. Object race detection. In *ACM Conference on Object-Oriented Programming Systems Languages, and Applications (OOPSLA)*, Oct. 2001.
- [37] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *The 22nd Annual International Symposium on Computer Architecture (ISCA '92)*, June 1995.
- [38] O. Zeev-Ben-Mordehai. Efficient integration of on-the-fly data race detection in distributed shared memory and symmetric multiprocessor environments. Master's thesis, Technion - Israel Institute of Technology, 2001.