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

Extended Abstract[*]

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

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

Abstract

Data race detection is highly essential for debugging multithreaded programs and assuring their correctness. Nevertheless, there is no single universal technique capable of handling the task efficiently, since the data race detection problem is computationally hard in the general case. Thus, all currently available tools, when applied to some general case program, usually result in excessive false alarms or in a large number of undetected races.

Another major drawback of 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.

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, gives highlights of its implementation issues, and suggests some optimizations. It shows that the overheads imposed by MultiRace are often much smaller (orders of magnitude) than those obtained by other existing tools.

[*]The full version of the paper can be found at http://www.cs.technion.ac.il/∼assaf/publications/MultiRace.pdf

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, 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) [14]. These are the real races that might happen in any specific execution of a program. 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* [14]. 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 [13] that apparent races exist if and only if at least one feasible race exists somewhere in the execution. Yet, according to same paper, the problem of exhaustively locating all apparent data races is still NP-hard.

Methods for detecting apparent data races use one of two main approaches – either *static* or *dynamic* detection. Sometimes a combination of these approaches is used for additional efficiency and accuracy. The static methods [3, 6, 7, 12] perform a compile-time analysis of the program's source code. In contrast, the dynamic methods, which are further divided into *postmortem* [2, 13] and *on-the-fly* [5, 15, 16] techniques, use some tracing mechanism to detect whether a particular execution of the program actually exhibited data races. The on-the-fly methods, which are of our main interest in this paper, buffer partial trace information in memory, analyze it, and detect data races as they occur. Thus, some of the on-the-fly methods are able to pinpoint the exact locations of the instructions that are involved in a detected race.

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

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

Finally, the issue of *detection granularity* is also of utmost importance: a small unit results in missed data races and higher overheads, whereas a large one causes false detection. In modern object-oriented programming languages, objects are the natural choice for becoming the units on which the detection should be applied: they usually contain data that is strongly related, and should be protected as a whole. This is the reason why the locking granularity in modern languages such as Java (as realized in the *monitor* APIs) is applied in granularity of objects[1]. We remark that the average object size is known to be very small (about 20 bytes for Java objects), hence the average detection accuracy on objects is relatively good. Unfortunately, all the currently known dynamic methods are limited to detection granularity of fixed size, 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.

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

In this paper we present a novel testing tool called MultiRace, which combines two very powerful techniques for on-the-fly detection of apparent data races. The first technique is a revised version of the Djit algorithm [9], called Djit$^+$. It is capable of efficiently detecting apparent races as they occur. The second technique is an improved variant of the Lockset algorithm [16], 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 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. These algorithms complement each other in that one compensates for the shortcomings of the other. The combination of both algorithms in one tool makes the detection of data races much more powerful.

MultiRace makes use of several novel technologies. By exploiting a unique configuration of memory mappings, called *views*, and the technique of *pointer swizzling* [4, 8], MultiRace detects data races in granularity of 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 detection mechanisms are injected in places where accesses to shared locations are performed.

Finally, 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 slowdowns reported in previous on-the-fly works are much higher for similar kinds of applications.

Because of its transparency, its relatively low overhead, its powerful detection algorithms, and its 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.

2 Detection Algorithms

We follow the basic observation that a synchronization object S can be *released* by one set of threads that reach a certain point in their execution and can then be *acquired* by another set of threads. For example, an `unlock(S)` operation releases S and a corresponding `lock(S)` operation acquires it; a global `barrier(S)` operation causes all threads to release S as they reach the barrier, and only then causes them to re-acquire S.

We base our definition of data races on the *happened-before* relation, denoted $\xrightarrow{hb}$, first suggested by Lamport in [10]. This definition also resembles those presented in [2, 9]; it allows recognition of apparent data races that pop up in some specific program execution.

Definition *There exists an apparent data race between two accesses to the same shared location, α and β, if α and β are executed by distinct threads, they are not synchronized (i.e., neither $\alpha \xrightarrow{hb} \beta$ nor $\beta \xrightarrow{hb} \alpha$), and at least one of them is for writing.*

2.1 Djit$^+$

In order to detect data races, we use an algorithm called Djit$^+$, which is a revised version of the earlier Djit algorithm [9]. The main disadvantages of the original Djit were the assumption of an underlying *sequentially consistent* system (which restricts the employment of many efficient optimizations) and the ability to detect only the very first data race in an execution. In contrast, Djit$^+$ can correctly operate on *weakly ordered* systems, such as the one presented by Adve and Hill in [1], and still detect a greater number of races as they occur in the program's execution.

Djit$^+$ relies on a formal framework called *vector time frames*, which is based on Mattern's virtual time vector time-stamps [11]. 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.

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

We assume that the maximum number of threads is known and stored in a constant called *maxThreads*. Each thread t maintains a vector of time frames, denoted $st_t[.]$, having *maxThreads* entries. For the sake of simplicity we assume that the IDs of the threads are positive integers in the range of [0,*maxThreads*-1]. For each index u, the entry $st_t[u]$ stores the latest local time frame of thread u, whose release operation is "known" to thread t. During the execution, the vectors of the threads are maintained in the following way: (1) if t performs a release, it increments its own time frame; (2) if u acquires a synchronization object 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.

In the full version of the paper we prove that the $\xrightarrow{hb}$ relation can be verified from the vectors of the time frames defined above. Thus, to detect data races on-the-fly it is sufficient to check the time frame vector of each newly logged access with the time frame vectors of all previously logged accesses. In the full version of the paper we also show that in order to improve performance and reduce overhead it is possible to restrict the logging and checking to only a small portion of all accesses. We prove that the algorithm remains correct if only the first read and the first write accesses to each of the shared locations in each time frame are logged. Furthermore, we prove that the algorithm remains correct even if the data race checks are performed only between the currently logged access and the most recently logged accesses in each of the other threads. Obviously, all these allow us to significantly reduce the overhead involved.

In order to implement the detection protocol, each thread t and each synchronization object S hold a vector of time frames. 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. Figure 1 shows the full Djit$^+$ protocol.

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

2.2 Lockset

To perform even more efficient detection of data races we also use a refined and optimized version of the Lockset algorithm, first presented in [16]. 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 barriers, so that it can be integrated with Djit$^+$ within the same framework (due to space limitations we do not show the extension in this paper).

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 test-

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 1. The full Djit$^+$ protocol

ing tool called Eraser [16], 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 in the execution so far. For each thread t, $locks\_held(t)$ holds at any given moment the set of all locks acquired by t. The algorithm itself, also called *lockset refinement*, is depicted in Figure 2.

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 $lh \leftarrow lh \cup \{reader\_lock\}$.
3. $C(v) \leftarrow C(v) \cap lh$.
4. If $C(v) = \emptyset$, then a race warning is issued.

Figure 2. The Lockset Algorithm

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

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

2.3 Benefits of Combining Lockset and Djit$^+$

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

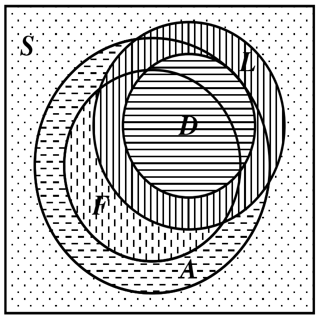

Figure 3. Correlation between sets of accessed locations

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

- In contrast to Djit$^+$, the Lockset algorithm was found to be quite insensitive to differences in thread interleavings, and it was shown to provide a certain kind of global information about the raced shared locations in a program. (However, despite this insensitivity, different sets of races might still appear when the program takes another execution path.)

- Since every data race is also a violation of the locking discipline, for many types of programs it can be said that Lockset and Djit$^+$ detect respectively a superset and a subset of all the raced shared locations in the execution. Therefore, it can be concluded that if Lockset did not produce any warnings in some execution, then there is a high probability that Djit$^+$ will not locate any additional races in further executions. Figure 3 demonstrates this observation: the set of all shared locations in some given program P is represented by region S; the set of shared locations that are participating in feasible data races in P is represented by region F; the set of shared locations that are participating in apparent data races in P is represented by region A, which is a superset of F; the set of shared locations for which Lockset detects violations of the locking discipline in some specific execution E_P of P is represented by region L; the set of shared locations that Djit$^+$ reports as participating in data races in E_P is represented by region D, which is always a subset of both A and L. We remark that for many types of programs (e.g., such that are not *completely nondeterministic* [6]), A becomes a subset of L.

- 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.

- Since both detection algorithms require same access logging mechanism, most of the overhead involved with the logging can be payed only once.

3 MultiRace—Implementation of the Logging Mechanism

In the following sections we describe MultiRace—the actual framework for implementing the logging mechanism and the data race detection algorithms.

As was mentioned before, 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 [8] and [4], which presented the concept of *views*. According to this concept, a physical memory page can be viewed from several virtual pages, called *views*, each having its own protection. Each object that resides on the corresponding physical page can be accessed through each of the different views. This attribute helps to distinguish between read and write accesses to the shared objects. In addition, this enables the realization of the variable-size detection unit, and thus avoids the fixed-size granularity problem usually faced by other data race detection tools. The concept of views and the memory layout imposed by it are depicted in Figure 4.

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

Modern operating systems allow a user handler function to be provided for different kinds of software and hardware exceptions. In the case of a page fault, the handler is supplied with the faulted memory address, the faulting instruction, the type of page fault (read or write), and the states of the machine registers. Once this information has been obtained, appropriate action can be taken: the faulted minipage can be located, its view tested and modified, and the race detection mechanisms invoked.

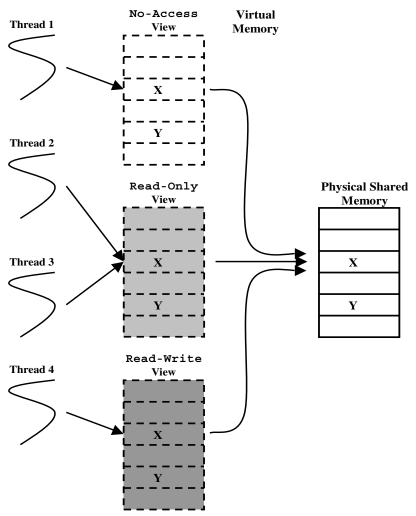

Figure 4. The memory layout and the depiction of views

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

In order to correctly activate the views swizzling and the data race detection mechanisms, the source code of the program is instrumented—memory allocation routines are overloaded, and all relevant accesses to global, static and dynamically allocated objects are intercepted and redirected to come from the correct view.

For the instrumentation to work properly, we require from the program code to be written in C++, and compile correctly prior to being changed. Under these conditions, we transparently perform the instrumentation with a simple automatic preprocessing tool. After the modifications are completed we guarantee that the program still compiles and runs correctly. For brevity's sake, we omit the description of the instrumentation process. The exact details appear in the full version of the paper.

4 Variable-Size Detection Unit

As was mentioned before, the detection unit in our implementation is dynamic. More precisely, we detect races in granularity of minipages and not in granularity of a fixed number of bytes. For this purpose, each minipage is associated with the information essential for the data race detection algorithms—the access history for Djit$^+$and the current state of 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. In fact, our instrumentation approach implies that the detection granularity is at least the size of an entire object defined by the classes and structures of the program. Since in all modern object-oriented programming languages the objects tend to be small and self-contained, consisting of only strongly related data fields, 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 one large object, with views swizzled for all the elements at once. Obviously, this also minimizes the additional space that is needed for the data race detection algorithms. It can, however, become a source for false alarms, when different threads access non-overlapping regions of the array.

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

Figure 5 demonstrates the reduction in slowdowns for a race-free version of the FFT application, when the granularity of source and destination matrices is changed from 1 complex number per minipage to 256. The overheads are calculated relatively to the original uninstrumented version with the corresponding number of threads. The figure also depicts the overheads for a situation in which each matrix is entirely located on a corresponding single minipage. In this case, the overheads drop sharply.

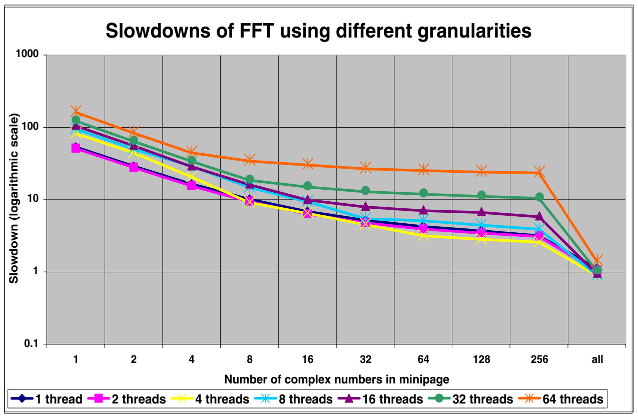

Figure 5. Overheads of a race-free FFT benchmark application with different granularities

5 Measured Overheads

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

	Shared Memory	# Mini-pages	Write/Read Faults	# Frames
FFT	3 MB	4	9/10	20
IS	128 KB	3	60/90	98
LU	8 MB	5	127/186	138
SOR	8 MB	2	202/200	206
TSP	1 MB	9	2792/3826	678
WATER	500 KB	3	15438/15720	15636

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

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

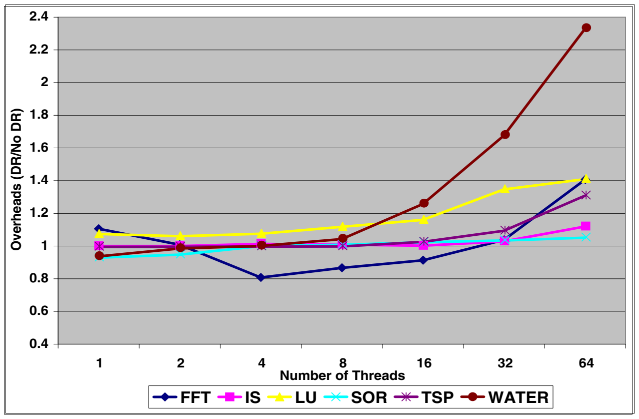

Figure 6. Overheads measured for the benchmarks using 1, 2, 4, 8, 16, 32 and 64 threads

Figure 6 presents the overheads obtained for our benchmark applications (without taking into account the time required to initialize the system). The overheads seem to be low and steady for 1–8 threads. The applications have either very low overheads or none at all. This suggests that our system is scalable in the number of CPUs. However, most of the applications suffer from heavier overheads for a higher number of threads. The reason is that the access logging and race detection mechanisms have to be activated separately for each of the running threads. In addition, more threads require more inter-thread communication. All this results in more time frames, more page faults, and thus more work performed by MultiRace per benchmark execution.

6 Conclusions

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

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

References

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

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

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

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

[5] A. Dinning and E. Schonberg. 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.

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

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

[8] 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.

[9] 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.

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

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

[12] 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.

[13] R. H. B. Netzer and B. P. Miller. Detecting data races in parallel program executions. Technical report, University of Wisconsin, Aug. 1990.

[14] 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.

[15] 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.

[16] 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.