Optimizing Threads Schedule Alignments to Expose the Interference Bug Pattern

Neelesh Bhattacharya[1], Olfat El-Mahi[1], Etienne Duclos[1], Giovanni Beltrame[1], Giuliano Antoniol[1], Sébastien Le Digabel[2] and Yann-Gaël Guéhéneuc[1]

[1] Department of Computer and Software Engineering
[2] GERAD and Department of Mathematics and Industrial Engineering
École Polytechnique de Montréal, Québec, Canada
(neelesh.bhattacharya,olfat.ibrahim,etienne.duclos,giovani.beltrame,giuliano.antoniol,yann-gael.gueheneuc)@polymtl.ca,
sebastien.le.digabel@gerad.ca

Abstract. Managing and controlling interference conditions in multi-threaded programs has been an issue of worry for application developers for a long time. Typically, when write events from two concurrent threads to the same shared variable are not properly protected, an occurrence of the interference bug pattern could be exposed. We propose a mathematical formulation and its resolution to maximize the possibility of exposing occurrences of the interference bug pattern. We formulate and solve the issue as an optimization problem that gives us (1) the optimal position to inject a delay in the execution flow of a thread and (2) the optimal duration for this delay to align at least two different write events in a multi-threaded program. To run the injected threads and calculate the thread execution times for validating the results, we use a virtual platform modelling a perfectly parallel system. All the effects due to the operating system's scheduler or the latencies of hardware components are reduced to zero, exposing only the interactions between threads. To the best of our knowledge, no previous work has formalized the alignment of memory access events to expose occurrences of the interference bug pattern. We use three different algorithms (random, stochastic hill climbing, and simulated annealing) to solve the optimization problem and compare their performance. We carry out experiments on four small synthetic programs and three real-world applications with varying numbers of threads and read/write executions. Our results show that the possibility of exposing interference bug pattern can be significantly enhanced, and that metaheuristics (hill climbing and simulated annealing) provide much better results than a random algorithm.

Keywords: Multi-Threaded Programs Testing, Optimization Techniques, Interference Bug Pattern.

1 Introduction

The advent of multi-core systems has greatly increased the use of concurrent programming. To control concurrent programs and their correct execution, the use of locks, semaphores, and barriers has become a common practice. However, despite of the use of locks and barriers, it is extremely difficult to detect and remove bugs, specifically data-race conditions and deadlocks. Most bugs are detected by testing, i.e. by multiple runs of a program under various environmental conditions. In fact, the same test case might or might not detect a bug because of a system's non-deterministic components (interrupts, scheduler, cache, etc.), over which the testers have no direct control.

The interference bug pattern is one the most common bugs in concurrent programs. For example, when two or more write events happen in very close proximity on unprotected shared data, the chances of incurring in an interference bug are high. This bug is also one of the hardest to eradicate [1, 2].

In this paper, we propose a theoretical formulation that helps maximizing the possibility of exposing interference bug pattern in multi-threaded programs, if it exists. An interference bug might occur when (1) two or more concurrent threads access a shared variable, (2) at least one access is a write, and (3) the threads use no explicit mechanism to enforce strict access ordering. To expose this bug, we want to inject a delay in the execution flow of each thread with the purpose of aligning in time different shared memory access events. Specifically, we want to align two write events occurring in two or more threads that share the same variable to maximize the probability of exposing an interference bug. We use unprotected variables (without locks or semaphores), so that bugs can occur and we can maximize the possibilities of finding them. Our formulation allows the identification of the optimal delays to be injected (positions and durations) using search or optimization algorithms. In particular, we use: random exploration, stochastic hill climbing, and simulated annealing.

We apply our approach to a set of multi-threaded data-sharing programs, called Programs Under Test (PUTs) that comprises of four synthetic programs and three real-world applications, CFFT (Continuous Fast Fourier Transform), CFFT6 (Continuous Fast Fourier Transform 6) and FFMPEG. CFFT computes the Fast Fourier Transform on an input signal, while CFFT6 performs a number of iterations of the Bailey's 6-step FFT to computes 1D FFT of an input signal. FFMPEG is a complete, cross-platform solution to record, convert and stream audio and video. To avoid non-determinism we use a simulation platform (ReSP [3]) which gives us full control over all the components of the system for building a fully-parallel execution environment for multi-threaded programs. We model an environment in which all the common limitations of a physical hardware platform (*i.e.*, the number of processors, memory bandwidth, and so on) are not present and all the the operating system's latencies are set to zero. The PUTs are executed in this environment, exposing only the threads' inherent interactions. We collect the exact times of each memory access event and then run, inject delays, and verify whether any interference bugs are exposed.

The rest of this paper is organized as follows: Section 2 presents the relevant previous work; Section 3 recalls some useful notions on interference bug patterns, meta-heuristics and optimization; Section 4 presents our formulation of the interference bug pattern and the detection approach; Section 5 describes the context and research questions of the experimental study; Section 6 reports our results while Section 7 discusses the trend observed in the results and specifies the threats to the validity of our results;

finally, Section 8 draws some concluding remarks and outlines the scope for future work.

2 Related Work

Our work seeks to maximize the possibility of exposing data-race and interference conditions. There exists significant works on the impact of data-race conditions in concurrent programs. Artho et al. [4] provided a higher abstraction level for data races to detect inconsistent uses of shared variables and moved a step ahead with a new notion of high-level data races that dealt with accesses to set of fields that are related, introducing concepts like *view* and *view consistency* to provide a notation for the new properties. However they focused on programs containing locks and other protections, while our work concerns unprotected programs.

Moving on to research carried out on bug patterns, Hovemeyer et al. [5] used bug pattern detectors to find correctness and perfomance-related bugs in several Java programs and found that there exists a significant class of easily detectable bugs. They were able to identify large number of bugs in real applications. But they faced the problem of knowing the actual population of real bugs in large programs. Farchi et al. [6] proposed a taxonomy for creating timing heuristics for the ConTest tool [7], showing it could be used to enhance the bug finding ability of the tool. They introduced the sleep, losing notify, and dead thread bug patterns. Eytani et al. [8] proposed a benchmark of programs containing multi-threaded bugs for developing testing tools. They asked undergraduates to create some buggy Java programs, and found that a number of these bugs cannot be uncovered by tools like ConTest [7] and raceFinder [9]. Bradbury et al. [10] proposed a set of mutation operators for concurrency programs used to mutate the portions of code responsible for concurrency to expose a large set of bugs, along with a list of fifteen common bug patterns in multi-threaded programs. Long et al. [11] proposed a method for verifying concurrent Java programs with static and dynamic tools and techniques using Petri nets for Java concurrency. They found proper verification tools for each failure. However all these approaches mainly focused on Java programs, so are language specific, while our approach is generic for every programming languages.

In the field of the behavior of concurrent programs, Carver et al. [12] proposed repeatable deterministic testing, while the idea of systematic generation of all thread schedules for concurrent program testing came with works on reachability testing [13, 14]. The VeriSoft model checker [15] applied state exploration directly to executable programs, enumerating states rather than schedules. ConTest [7] is a lightweight testing tool that uses various heuristics to create scheduling variance by inserting random delays in a multi-threaded program. CalFuzzer [16] and CTrigger [17] use analysis techniques to guide schedules toward potential concurrency errors, such as data races, deadlocks, and atomicity violations.

One of the most influential tools developed for testing concurrent programs is CHESS [18]. It overcomes most of the limitations of the tools developed before. What set CHESS apart from its predecessors is its focus on detecting both safety and liveness violations in large multi-threaded programs. It relies on effective safety and liveness testing of such programs, which requires novel techniques for preemption bounding and fair scheduling. It allows a greater control over thread scheduling than the other tools and, thus, provides higher-coverage and guarantees better reproducibility. CHESS tries all the possible schedules to find a bug, whereas we create the schedules that maximizes the likelihood of exposing an interference bug, if it is present.

In this paper, we do not intend to compare our approach with CHESS or other mentioned tools. We want to help developers by providing them with the locations and durations of delays to inject in their multi-threaded programs so that they can, subsequently, run their programs to enhance the likelihood of exposing interference bugs, possibly using CHESS. It is to be noted that our work clearly differs from the previous ones, in the sense that none of them played with inserted delays to align the write events ; they explore the various possible schedules to expose a bug.

3 Background Notions

Concurrency is built around the notion of multi-threaded programs. A thread is defined as an execution context or a lightweight process having a single sequential flow of control within a program [19]. Inserting delays in the execution of a thread is an efficient way of disrupting its normal behavior: the inserted delay shifts the execution of the subsequent statements. By doing so, an event in one thread can be positioned in close proximity with another event in another thread, increasing the probability of exposing an interference bug.

Bradbury et al. [10] mentioned fifteen possible bug patterns that could affect the normal behavior of threads and cause severe problems to concurrency. Out of them, we considered the interference bug pattern because it is one of the most commonly encountered and one of the hardest to eradicate [1, 2].

3.1 Interference Bug Sequence Diagrams

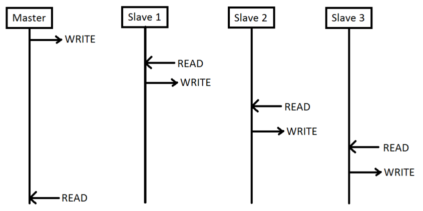

Fig. 1. Behavior of a PUT

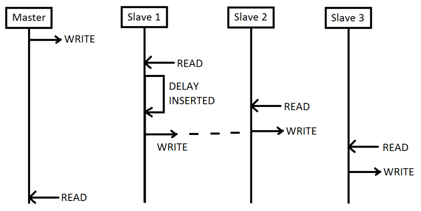

Fig. 2. PUT with injected delay

In Figures 1 and 2, a master thread creates child threads and shares some data with them. The expected behavior (without the bug), illustrated in Figure 1, is that each child thread accesses the shared data sequentially, so that every thread has the last version of the data when it reads it. When we inject a delay just before a child writes its modification to the shared data, as shown in Figure 2, another thread reads a wrong datum and may produce incorrect results.

3.2 Search Algorithms

Given the number of possible thread events in any multi-threaded program, we apply search algorithms to maximize the number of "alignments" between events, *i.e.*, the number of events in close proximity. To experiment with different optimization techniques we chose the two most commoly used optimization algorithms: stochastic hill climbing (SHC) [20] and simulated annealing (SA) [21], and we validated them against random search (RND).

4 Formulation and Approach

Given a multi-threaded program, the interleaving of threads depends on the hardware (*e.g.*, number of processors, memory architecture, etc.) and the operating system. There are as many schedules as there are environmental conditions, schedule strategies, and policies of the operating system when handling threads. Among these schedules, there could be a subset leading to the expression of one or more interferences. In general, the exhaustive enumeration of all possible schedules is infeasible, and in an ideal situation, all threads would run in parallel.

4.1 Parallel Execution Environment

To provide a deterministic parallel execution environment without external influences we use a virtual platform, namely ReSP [3]. ReSP is a virtual environment for modeling an ideal multi-processor system with as many processors as there are threads. ReSP is also a platform based on an event-driven simulator that can model any multi-processor architecture and that implements an emulation layer that allows the interception of any OS call. ReSP allows access to all execution details, including time of operations in milliseconds, accessed memory locations, thread identifiers, and so on.

To create our parallel execution environment, we model a system as a collection of Processing Elements (PEs), ARM cores in our specific case but any other processor architecture could be used, directly connected to a single shared memory, as shown in Figure 3. Therefore, our environment makes as many PEs available as there are execution threads in a PUT, each thread being mapped to a single PE. PEs have no cache memory, their interconnection is implemented by a 0-latency crossbar, and the memory responds instantaneously, thus each thread can run unimpeded by the sharing of hardware resources. This environment corresponds to an ideal situation in which the fastest possible execution is obtained.

We use ReSP to run our unmodified test cases without (except when injecting delays) and to calculate the threads' execution time for validating our results. We use ReSP's ability to trap any function call being executed on the PEs to route all OS-related activities outside the virtual environment. Thread-management calls and other OS functions (`sbrk`, file access, and so on) are handled by the host environment, without affecting the timing behaviour of the multi-threaded program. Thus, the PUT perceives that all OS functions are executed instantaneously without access to shared resources. Because all OS functions are trapped, there is no need for a real OS implementation to run the PUT in the virtual environment. This is to say the PUTs are run without any external interference. The component performing the routing of OS functions, referred to as the OS Emulator, takes care of processor initialization (registers, memory allocation, and so on) and implements a FIFO scheduler that assigns each thread to a free PE as soon as it is created.

The main difference between our virtual environment and a real computer system are cache effects, the limited number of cores, other applications running concurrently on the same hardware and other interactions that make scheduling non-deterministic and introduce extra times between events. In our environment, the full parallelism of the PUT is exposed and the only interactions left are inherent to the PUT itself.

4.2 Problem Formalization

Any real hardware/software environment will deviate from the parallel execution environment described above. From the threads' point of view, any deviation will result in one or more delays inserted in their fully-parallel delay-free execution. Figure 4 summarizes the execution of four threads, where each thread performs four read and–or write accesses. For the sake of simplicity, let us assume that just one delay is inserted in a given thread. This delay will shift the thread's computation forward in time, and possibly cause some memory write access(es) to happen in close proximity, leading to the possibility of exposing an interference condition.

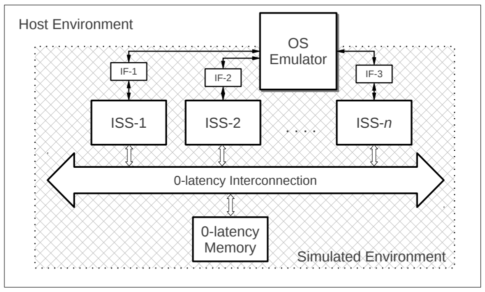

Fig. 3. The virtual platform on which the PUT is mapped: each component except for the processors has zero latency, and OS calls are trapped and executed externally

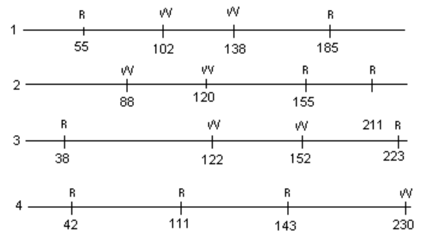

Fig. 4. Concurrent Threads Example

Concretely, at 102 ms the first thread writes into a memory location a value and, given the schedule, no interference happens. In other words, two threads do not attempt a write at the same time in the same memory location. However, if, for any reason, the thread schedule is different and, for example, thread 2 is delayed by 2 ms after the first writing then a possible interference happens at 122 ms between threads 2 and 3.

The event schedule depends on the operating system, the computer workload, other concurrent programs being run, and the scheduler policy. Therefore, enhancing the possibility of exposing an interference bug via testing for any foreseeable schedule is a challenging problem that has been addressed in several ways, from search-based approaches [22, 23] to formal methods [24, 25]. The higher the number of available CPUs, the higher the number of scheduled threads and events, and the more difficult it is to manually verify that any two threads will not create an interference under any possible schedule. The example in figure 4 shows a larger system, *i.e.*, with a higher number of threads and a longer execution time. Also in this case, the delays would be inserted in a similar manner between events (taking into account the longer execution time). In general, we believe that our approach would be able to increase the chances of exposing the interference conditions for systems of any size.

Let the PUT be composed on N threads. Let us assume that the i^{th} thread contains M_i events; let $t_{i,j}$ be, for the thread i, the time at which an event (*e.g.*, a memory access, a function call, and so on) happens and let i^* be the thread subject to perturbation, *i.e.*, the thread in which a delay $\varDelta$ will be injected before an event p. Finally, let $a_{i,j}$ stands for the action performed at time $t_{i,j}$ by the thread i. Because ReSP allows to precisely track memory accesses as well as times, to simplify the formalization, let us further assume that $a_{i,j}$ equals to 1 to show that it is a "write" action to a given memory cell or 0 to show some other action. Then, our objective is to maximize the number of possible interferences $N_{Interference}$:

$$N_{Interference} = \max_{\varDelta,p,i*} \left\{ \sum_{i=1,i\neq i*}^{N} \sum_{j=1}^{M_i} \sum_{k=p}^{M_{i*}} \delta(a_{i*,k}, a_{i,j}) \delta(t_{i,j}, t_{i^*,k} + \varDelta) \right\} \quad (1)$$

under the constraint $t_{i,j} \geq t_{i^*,k} + \varDelta$, and where $\delta(x, y)$ is the Kronecker operator[3].

We want to maximize the numbers of alignments, *i.e.*, two write events coinciding at the same time occurring in the same memory location, using Equation 1. Unfortunately, this equation leads to a staircase-like landscape as it result in a sum of 0 or 1. Any search strategy will have poor guidance with this fitness function and chances are that it will behave akin to a random search.

If we assume that a delay is inserted before each write event in all threads, then all threads events will be shifted. More precisely, if $\varDelta_{i,j}$ is the delay inserted between the events $j-1$ and j of the thread i, all times after $t_{i,j}$ will be shifted. This shift leads to new time τ for the event $a_{i,j}$:

$$\tau_{i,j}(a_{i,j}) = t_{i,j} + \sum_{k=1}^{j} \varDelta_{i,k} \quad (2)$$

[3] The Kronecker operator is a function of two variables, usually integers, which is 1 if they are equal and 0 otherwise ($\delta(x, y) = 1$, if $x = y$; or 0 otherwise).

Considering the difference between $\tau_{i_q,j_q}(a_{i_q,j_q})$ and $\tau_{i_r,j_r}(a_{i_r,j_r})$, when both a_{i_q,j_q} and a_{i_r,j_r} are write events to the same memory location, we rewrite Equation 1 as:

$$N_{Interference}(write) = \max_{\Delta_{1,1},\ldots,\Delta_{N,N_N}} \left\{ \sum_{i_r}^{N} \sum_{j_r}^{M_{j_r}} \right.$$
$$\left. \sum_{i_q \neq i_r}^{N} \sum_{j_q}^{M_{j_q}} \frac{1}{1 + |\tau_{i_q,j_q}(a_{i_q,j_q}) - \tau_{i_r,j_r}(a_{i_r,j_r})|} \right\} \tag{3}$$

under the constraints $\tau_{i_q,j_q} \geq \tau_{i_r,j_r}$ and $a_{i_q,j_q} = a_{i_r,j_r} = write$ to the same memory location.

Equation 3 leads to a minimization problem:

$$N_{Interference}(write) = \min_{\Delta_{1,1},\ldots,\Delta_{N,N_N}} \left\{ \sum_{i_r}^{N} \sum_{j_r}^{M_{j_r}} \right.$$
$$\left. \sum_{i_q \neq i_r}^{N} \sum_{j_q}^{M_{j_q}} (1 - \frac{1}{1 + |\tau_{i_q,j_q}(a_{i_q,j_q}) - \tau_{i_r,j_r}(a_{i_r,j_r})|}) \right\} \tag{4}$$

For both Equations 3 and 4, given a $\tau_{i_q,j_q}(a_{i_q,j_q})$, we may restrict the search to the closest event in the other threads, typically: $\tau_{i_r,j_r}(a_{i_r,j_r}) \geq \tau_{i_q,j_q}(a_{i_q,j_q})$ & $\tau_{i_q,j_q}(a_{i_q,j_q}) \leq \tau_{i_q,j_s}(a_{i_q,j_s})$. Under this restriction, **Equation 4 is the fitness function used in the search algorithms to inject appropriate delays in threads to maximize the probability of exposing interference bugs (if any).** This equation also solves the staircase-like landscape problem because the fitness function is sum of real numbers, providing a smoother landscape.

4.3 Problem Modeling and Settings

As described above, the key concepts in our thread interference model are the times and types of thread events. To assess the feasibility of modeling thread interferences by mimicking an ideal execution environment, we are considering simple problem configurations. More complex cases will be considered in future works; for example, modeling different communication mechanisms, such as pipes, queues, or sockets. We do not explicitly model resource-locking mechanisms (*e.g.*, semaphores, barriers, etc.) as they are used to protect data and would simply enforce a particular event ordering. Therefore, if data is properly protected, we would simply fail to align two particular events. At this stage, we are also not interested in exposing deadlock or starvation bugs.

From Equations 2 and 4, we can model the problem using the times and types of thread events. Once the occurrence write events has been timestamped using ReSP, we can model different schedules by shifting events forward in time. In practice, for a given initial schedule, all possible thread schedules are obtained by adding delays between thread events (using Equation 2).

Our fitness function 4 (*i.e.*,what an expression of how fit is our schedule to help expose interference bugs) can be computed iterating over all thread write events. The fitness computation has therefore a quadratic cost over the total number of write events. However, for a given write event, only events occurring in times greater or equal to the current write are of interest, thus making the fitness evaluation faster (though still quadratic in theory).

In general, it may be difficult or impossible to know the maximum delay that a given thread will experience. Once a thread is suspended, other threads of the same program (or other programs) will be executed. As our PUTs are executed in an ideal environment (no time sharing, preemption, priority, and so on), there is no need to model the scheduling policy and we can freely insert any delay at any location in the system to increase the chances of exposing the interference bug.

5 Empirical Study

The *goal* of our empirical study is to obtain the conceptual proof of the feasibility and the effectiveness of our search-based interference detection approach and validate our fitness function (see Equation 4).

As a metric for the quality of our model, we take the number of times we succeed in aligning two different write events to the same memory location. Such alignment increases the chances of exposing an interference bug, and can be used by developers to identify where data is not properly protected in the code.

To perform our conceptual proof, we have investigated the following two research questions:
RQ1: Can our approach be effectively and efficiently used on simple as well as real-world programs to maximize the probability of interferences between threads?
RQ2: How does the dimension of the search space impact the performance of three search algorithms: RND, SHC and SA?

The first research question aims at verifying that our fitness function guides the search appropriately, leading to convergence in an acceptable amount of time. The second research question concerns the choice of the search algorithm to maximize the probability of exposing interference bugs. The need for search algorithms is verified by comparing SHC and SA with a simple random search (RND): better performance from SHC and SA increases our confidence in the appropriateness of our fitness function.

One advantage of this approach is that it has no false positives, because it doesn't introduce any functional modification in the code: if a bug is exposed, the data are effectively unprotected. Nevertheless, we do not guarantee that a bug will be exposed even if present, as the approach simply increases the likelihood of showing interferences. The chances of exposure are increased because the manifestation of interference bug depends on the thread schedule and we, unlike other approaches, manufacture the schedules that maximize interference.

It might be argued that a data race detector does not need the timings of write alignments to be so accurate. But as we are dealing with a fully parallelized environment, we target specific instances and align the events with much more precision than what required by a data race detector [4]. Basically, we are pin-pointing the event times with accuracy.

Our approach can also be used in cases where it is enough to have two change the order of two events to verify the correctness of some code. Once the events are aligned, an arbitrarily small additional delay would change the order of two events, possibly exposing data protection issues. The correct use of locks or other data protection measures would prevent a change in the order of the events.

6 Experimental Results

Table 1. Application Details

PUTs	LOCs	Nbr. of Threads	Events
Matrix Multiplication (MM)	215	4	RWWR, WWWW, RRWW, WWRR
Count Shared Data (CSD)	160	4	WWW, RRW, RWRW, RW
Average of Numbers (AvN)	136	3	W, RW, RW
Area of Circle (AC)	237	5	RW, RWW, RRW, RWRW, RWWRW
CFFT	260	3	WR, WR, WRWRWR
CFFT6	535	3	WRWRWRWRWRWR, WR, WRWRWRWR
FFMPEG	2.9×10^5	4	WRWRRR, WR, WR, WR

Table 2. Execution Times for Real-World Applications, in milliseconds

	CFFT		CFFT6		FFMPEG	
	1×10^6	1×10^7	1×10^6	1×10^7	1×10^6	1×10^7
SA	**3118**	**5224**	**27443**	**20416**	**1562**	**4672**
HC	**3578**	**4976**	**27328**	**21943**	**1378**	**5100**
RND	**113521**	**107586**	**342523**	**339951**	**59599**	**133345**

To answer our two research questions, we implemented four small synthetic and three real-world multi-threaded programs with different numbers of threads and read/write events. Table 1 provides the details of these applications: their names, sizes in numbers of lines of code, number of threads, and sequences of read and write access into memory. The "Events" column shows the various events with a comma separating each thread. For example, the thread events in column 4 for row 3 (*Average of Numbers*) should be read as follows: Thread 1 has just one write event, thread 2 has a read, followed by a write event, and thread 3 has a read followed by a write event.

We believe that our results are independent of the execution system or architecture: our approach aligns write events among threads on a virtual system, and exposes data protection issues regardless of the final execution platform. Similarly, when applied to applications of any size, the interference conditions are exposed irrespective of the number of lines of code that the application may contain.

6.1 RQ1: Approach

RQ1 aims at verifying if our approach can effectively identify time events configurations, and thus schedules, leading to possible thread interferences. We experimented with RND, SHC, and SA. RND is used as a sanity check and to show that a search algorithm is effectively needed.

For the three search algorithms (RND, SHC and SA), we perform no more than 10^6 fitness evaluations. For the three algorithms, we draw the added delays $\Delta_{i,k}$ from a uniform-random distribution, between zero and a maximum admissible delay fixed to 10^7 ms (10^4 seconds).

We configured RND in such a way that we generated at most 10^7 random delays $\Delta_{i,k}$, uniformly distributed in the search space, and applied Equation 2 to compute the actual thread time events. We then evaluated each generated schedule, to check for interferences, using Equation 4.

We set the SHC restart value at 150 trials and we implemented a simple neighbor variable step visiting strategy. Our SHC uses RND to initialize its starting point, then it first attempts to move with a large step, two/three orders of magnitude smaller than the search space dimension, for 1000 times, finally it reduces the local search span by reducing the distance from its current solution to the next visited one. The step is drawn from a uniformly-distributed random variable. Thus, for a search space of 10^7, we first attempt to move with a maximum step of 10^4 for 20 times. If we fail to find a better solution, perhaps the optimum is close, and then we reduce the step to 10^3 for another 20 trials, and then to 500 and then to 50. Finally, if we do not improve in 150 move attempts (for the large search space), the search is discarded, a next starting solution is generated and the process restarted from the new initial solution. We selected the values and the heuristic encoded in the SHC via a process of trial and error. Intuitively, if the search space is large, we would like to sample distant regions but the step to reach a distant region is a function of the search space size, so we arbitrarily set the maximum step of two or three orders of magnitude smaller than the size of the search space.

We configured the SA algorithm similarly to the SHC algorithm, except for the cooling factor and maximum temperature. In our experiments, we set $r = 0.95$ and T_{max} depending on the search space, *i.e.*, 0.0001 for a search space of size 110^7 and 0.01 for smaller search spaces.

Regarding the times to compute the solutions with different algorithms, it is a well known fact that SHC and SA scale less than linearly with the design space. This fact can be proven by having a look at the results provided in table 2. Table 2 shows the execution times of various algorithms for computing the alignments 100 times only for the real-world applications with large search space (10^6 and 10^7 ms). It can be seen that despite the increasing size of the design space, SHC and SA converge to solutions in reasonable amounts of times, as compared to RND.

Figure 5 reports the relative performance over 100 experiments of RND, SHC, and SA for *Matrix Multiplication*, CFFT6 and FFMPEG for a search space of 10^7 ms. Each time that our approach has exposed a possible interference, we recorded the number of fitness evaluations and stopped the search. We obtained similar box-plots for the other applications, but do not show them here because they do not bring additional/contrasting insight in the behavior of our approach. The box-plots show that SHC outperforms RND and that SA and SHC perform similarly. However, SA performs marginally better than SHC The missing plots for CFFT6 in Figure 5 are due to the fact that RND did not find any solution in 10^6 iterations, even after 100 runs.

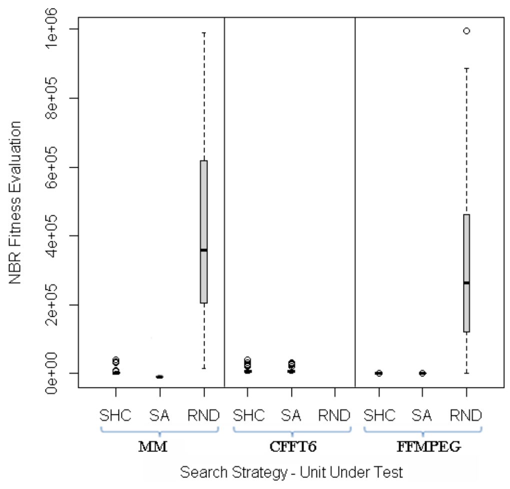

Fig. 5. (RQ1) Algorithm comparison for a search space up to 10 Million sec delay

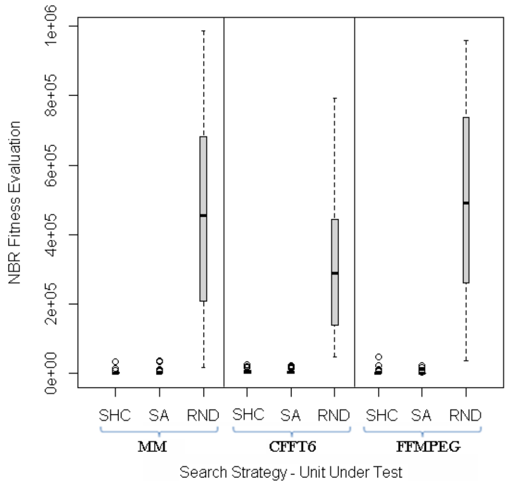

Fig. 6. (RQ2) Algorithm comparison for a search space up to 1 Million sec delay

Overall, we can positively answer our first research question: **our approach can effectively and efficiently be used on simple program models to maximize the possibility of interferences between threads.**

6.2 RQ2: Search Strategies

RQ2 aims at investigating the performance of the different strategies for various search-space dimensions. Our intuition is that, if the search space is not large, then even a random algorithm could be used. We set the maximum expected delay to 1 sec (10^3 ms), 10 sec (10^4 ms) and 1000 sec (10^6 ms). As the search space is smaller than that in RQ1, we also reduced the number of attempts to improve a solution before a restart for both SHC and SA as a compromise between exploring a local area and moving to a different search region. We set this number to 50.

Figures 6, Figure 7 and Figure 8 report the box-plots comparing the relative performance of RND, SHC, and SA for the first three synthetic and two real-world applications. Figure 7 has been made more readable by removing a single outlier value of 28,697 for the number of fitness evaluations of *Count Shared Data* when using SHC.

As expected, when the search space is small, RND performs comparably to both SHC and SA, as shown in Figure 7. However, when the search space size increases, SHC and SA perform better than RND, as shown in Figure 8 and 6.

One might argue that in Figure 7 *Average of Numbers* shows instances where some outliers for which SHC reaches almost the maximum number of iterations to align the events, which does not seem to be the case with RND. The explanation is that in small search spaces RND can perform as well as any other optimization algorithm, sometimes

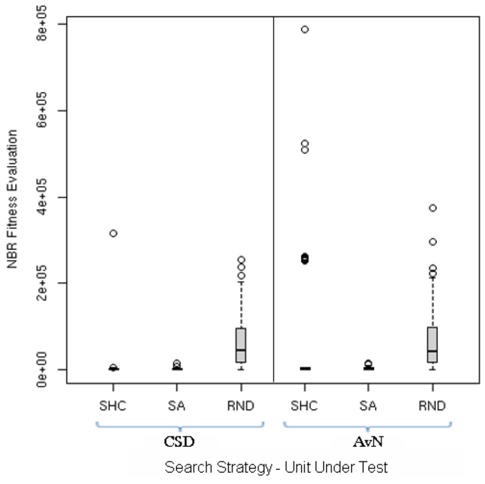

Fig. 7. (RQ2) Algorithm comparison for a search space up to 1 sec delay

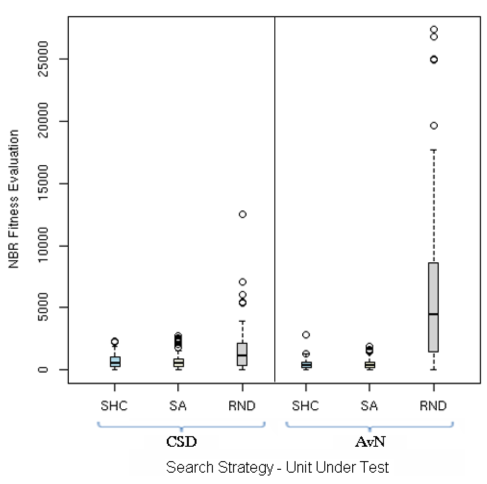

Fig. 8. (RQ2) Algorithm comparison for a search space up to 10 sec delay

even better. It is worth noting that even in a search space of 10^3 ms, there were instances where RND could not find a solution even within the maximum number of iterations (*i.e.*, 10^6 random solutions). SHC and SA were successful in exposing a possible bug each and every time (in some cases with higher number of iterations, which resulted in the outliers).

We also observe differences between the box-plots for the *Count Shared Data* and *Average of Numbers*. We explain this observation by the fact that *Count Shared Data* contains more write actions (see Table 1) than *Average of Numbers*. In other words, it is relatively easier to align two events in *Count Shared Data* than in *Average of Numbers*. Indeed, there are only three possible ways to create an interference in *Average of Numbers* while *Count Shared Data* has 17 different possibilities, a six-fold increase which is reflected into the results of Figure 7.

Once the search space size increases as in Figure 8, RND is outperformed by SHC and SA. In general, SA tends to perform better across all PUTs.

Overall, we can answer our second research question as follows: **the dimension of the search space impacts the performance of the three search algorithms. SA performs the best for the all PUTs and different delays.**

7 Discussion and Threats

Our results support the conceptual proof of the feasibility and the effectiveness of our search-based interference detection approach. They also show that our fitness function (Equation 4) is appropriate, as well as the usefulness of a virtual environment to enhance the probability of exposing interference bugs.

Exposed interferences are somehow artificial in nature as they are computed with respect to an ideal parallel execution environment. In fact, the identified schedules may not be feasible at all. This is not an issue, as we are trying to discover unprotected data accesses, and even if a bug is found with an unrealistic schedule, nothing prevents from being triggered by a different, feasible schedule. Making sure that shared data is properly protected makes code safer and more robust.

Although encouraging, our results are limited in their validity as our sample size includes only four small artificial and three real-world programs. This is a threat to *construct validity* concerning the relationship between theory and observations. To overcome this threat, we plan to apply our approach on more number of real-world programs in future work.

A threat to *internal validity* concerns the fact that, among the four artificial programs used, we developed three of them. However, they were developed long before we started this work by one of the authors to test the ReSP environment. Thus, they cannot be biased towards exposing interference bugs.

A threat to *external validity* involves the generalization of our results. The number of evaluated programs is small (a total of seven programs). Some of them are artificial, meant to be used for a proof of concept. Future work includes applying our approach to other large, real-world programs.

8 Conclusion

Detecting thread interference in multi-threaded programs is a difficult task as interference depends not only on the source code of the programs but also on the scheduler strategy, the workload of the CPUs, and the presence of other programs.

In this work, we proposed a novel approach based on running the programs under test on an ideal virtual platform, maximizing concurrency and inserting delays to maximize the likelihood of exposing interference between threads.

We used our fitness function and the three search algorithms to find the optimal delays on four small artificial and three real-world small/large multi-threaded programs. Our results show that our approach is viable and requires appropriate search strategies, as a simple random search won't find a solution in a reasonable amount of time.

Future work will be devoted to extending and enriching our interference model with more complex data structures such as pipes, shared memories or sockets.

References

1. Park, A.: Multithreaded programming (pthreads tutorial). `http://randu.org/tutorials/threads/` (1999)
2. Software Quality Research Group, Ontario Institute of Technology: Concurrency anti-pattern catalog for java. `http://faculty.uoit.ca/bradbury/concurr-catalog/` (2010)
3. Beltrame, G., Fossati, L., Sciuto, D.: ReSP: a nonintrusive transaction-level reflective MP-SoC simulation platform for design space exploration. Computer-Aided Design of Integrated Circuits and Systems (2009) 28–40

4. Artho, C., Havelund, K., Biere, A., Biere, A.: High-level data races. In: Journal on Software Testing, Verification & Reliability (STVR). (2003)
5. Hovemeyer, D., Pugh, W.: Finding bugs is easy. In: ACM SIGPLAN Notices. (2004) 132–136
6. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: Proceedings of the 17th International Symposium on Parallel and Distributed Processing. (2003) 286.2–
7. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for testing multi-threaded java programs. Concurrency and Computation: Practice and Experience **15**(3-5) (2003) 485–499
8. Eytani, Y., Ur, S.: Compiling a benchmark of documented multi-threaded bugs. Parallel and Distributed Processing Symposium, International **17** (2004) 266a
9. Ben-Asher, Y., Farchi, E., Eytani, Y.: Heuristics for finding concurrent bugs. In: Proceedings of the 17th International Symposium on Parallel and Distributed Processing. (2003) 288.1–
10. Bradbury, J.S., Cordy, J.R., Dingel, J.: Mutation operators for concurrent java (j2se 5.0) 1
11. Long, B., Strooper, P., Wildman, L.: A method for verifying concurrent java components based on an analysis of concurrency failures: Research articles. Concurr. Comput. : Pract. Exper. **19** (2007) 281–294
12. Carver, R.H., Tai, K.C.: Replay and testing for concurrent programs. IEEE Softw. **8** (1991) 66–74
13. Hwang, G.H., chung Tai, K., lu Huang, T.: Reachability testing: An approach to testing concurrent software. International Journal of Software Engineering and Knowledge Engineering **5** (1995) 493–510
14. Lei, Y., Carver, R.H.: Reachability testing of concurrent programs. IEEE Trans. Softw. Eng. **32** (2006) 382–403
15. Godefroid, P.: Model checking for programming languages using verisoft. In: In Proceedings of the 24th ACM Symposium on Principles of Programming Languages. (1997) 174–186
16. Joshi, P., Naik, M., seo Park, C., Sen, K.: Calfuzzer: An extensible active testing framework for concurrent programs
17. Park, S., Lu, S., Zhou, Y.: Ctrigger: exposing atomicity violation bugs from their hiding places. SIGPLAN Not. **44** (2009) 25–36
18. Musuvathi, M., Qadeer, S., Ball, T.: Chess: A systematic testing tool for concurrent software (2007)
19. Drake, D.G., JavaWorld.com: A quick tutorial on how to implement threads in java. `http://www.javaworld.com/javaworld/jw-04-1996/jw-04-threads.html` (1996)
20. Wattenberg, M., Juels, A.: Stochastic hillclimbing as a baseline method for evaluating genetic algorithms. In: Proceedings of the 1995 conference. Volume 8., Kaufmann (1996) 430
21. Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., Teller, E.: Equation of state calculations by fast computing machines. Journal of Chemical Physics **21** (1953) 1087–1092
22. Briand, L.C., Labiche, Y., Shousha, M.: Stress Testing Real-Time Systems with Genetic Algorithms. In: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO '05). (2005) 1021–1028
23. Briand, L.C., Labiche, Y., Shousha, M.: Using Genetic Algorithms for Early Schedulability Analysis and Stress Testing in Real-Time Systems. Genetic Programming and Evolvable Machines **7** (2006) 145–170
24. Flanagan, C., Freund, S.N.: Atomizer: A dynamic atomicity checker for multithreaded programs. Scientific Computer Program **71**(2) (2008) 89–109
25. Tripakis, S., Stergiou, C., Lublinerman, R.: Checking non-interference in spmd programs. In: 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar 2010). (June 2010) 1–6