SICS Technical Report T2007:02 ISRN:SICS-T–2007/02-SE
ISSN 1100-3154

Entropy injection

by

Lars Albertsson
2006-08-25
lalle@sics.se
Swedish Institute of Computer Science
Box 1263, S-164 29 Kista, Sweden

Abstract:

Testing is the predominant software quality assurance method today, but it has a major flaw — it cannot reliably catch race conditions, intermittent errors caused by factors that cannot be controlled during testing, such as unpredictable timing behaviour in concurrent software.

We present *entropy injection*, a extension of traditional test methods, which enable developers to create tests for arbitrary types of race conditions in any software application, reusing the application's existing test cases. An entropy injector runs the software under test in an instruction set simulator, where all factors that normally are unpredictable can be explicitly controlled. The injector provokes race condition defects by artificially changing the timing behaviour of the simulated processors, hardware devices, clocks, and input models. Provoked defects can be debugged by developers in a non-intrusive, programmable debugger, which allows race condition defects to be reproduced and provides access to all software state in a distributed system. Developers can use its services to create application-specific injection strategies and directed regression test cases that monitor application state and test specific interleavings of events.

Our proof-of-concept entropy injector implementation Njord is built on Nornir, a debugger environment based on the complete system simulator Simics. Njord provokes test case failures by suspending simulated processors, thereby injecting delays in the processes in a concurrent application. We demonstrate Njord on a small test routine, and show how a developer can write a race condition regression test that triggers errors with very high probability, or provoke errors with good probability without using application knowledge.

Keywords: entropy injection, noise making, race condition, complete system simulation, full system simulation, Nornir

1 Introduction

In the early days of computing, almost all programs were functional — the programs read data, performed a computation, and produced output data. Program output depended only on the input data. We call this functional behaviour or determinism, and it implies that program executions are reproducible. It is a very convenient property from a quality assurance perspective. If a developer can assume that a certain input data set results in the same output data for every execution, he only needs to execute a certain test scenario once in order to ensure that the scenario will work properly when the program is put in production. Reproducibility is also an unusual property in engineering. In other engineering disciplines, one cannot safely assume that a particular test execution is representative for all test executions with a certain input. The result of running a product test of a physical product or machine is also determined by factors that cannot be completely controlled, such as temperature, moisture, material fatigue, exact timing, fuel quality, corrosion, etc.

Reproducibility allows software developers to use test case based testing with automated regression test suites as the primary defect detection method. When a defect has been detected, the developer debugs the failing scenario and understand the defect in order to correct it. The execution of a functional computer program can usually be observed without affecting the result, e.g. by using a symbolic debugger, or inserting print statements. This is referred to as non-intrusive debugging/observation, or absence of probe-effect. If a defect in a deterministic program can be non-intrusively observed, it can be debugged by repeating the program execution, each time obtaining more information about the erroneous behaviour. Eventually, the developer will trace the error symptoms back to the error source. This procedure is called cyclic debugging.

Regression testing in combination with cyclic debugging is cost-effective, in comparison to other quality assurance methods in engineering. It is therefore the predominant software quality assurance method today.

Software has changed, however, and most programs today are nondeterministic and cannot be fully observed and debugged without affecting the execution. Traditional test methods allow test engineers to control most factors that affect execution, such as input data and hardware and software environment configuration. Some factors, for example process scheduling order and input arrival times, are typically outside the testers' control. Software applications therefore have intermittent defects caused by such uncontrollable execution factors. The most common type of intermittent defect is *race conditions*, defects dependent on uncontrollable timing factors. Since we have no cost-effective method for detecting race conditions, they are common in production software.

Race conditions are becoming increasingly common as computer systems become more concurrent and asynchronous. The drift towards more concurrent computer systems is gradual but inevitable. It started early in the history of computing, with the introduction of asynchronous hardware services, such as imprecise traps, and with reactive programs, which receive new input data during execution. Programs that receive interrupts or input data asynchronously

become sensitive to arrival times and ordering of interrupts and input data, and are therefore not deterministic. The nondeterminism in computer systems has continued to increase with the introduction of multitasking operating systems, multiprocessor machines, and networked computers. Recently, the processor vendors have unanimously concluded that it is not worthwhile to spend more efforts and silicon on improving single-thread performance. Instead, they put multiple processor cores on each chip. This change in technology will force software vendors that care about performance to write parallel software, and to convert their existing sequential software. Developing parallel software is inherently difficult, and race conditions are frequently introduced in the process. The proliferation of multicore processors will therefore drastically increase the need for test methods that are capable of detecting race conditions.

Most previous research in race condition detection has focused on detecting data races in homogeneous shared memory programs. There are tools that either analyse and report suspicious memory access behaviour, or control the behaviour of a single normally uncontrollable factor, e.g. process scheduling, and apply heuristics to explore new scheduling interleavings. Existing methods are able to detect some race conditions, but leave some factors uncontrolled, and do not achieve full reproducibility.

Entropy injection, described in this paper, is a simulation-based test method where *all* factors affecting test execution are explicitly controlled. Its main purpose is to provide means for developers to create directed regression test cases for detecting all classes of race condition defects. An entropy injector artificially changes timing and interleaving of events in the simulated system, and enables race condition defects to be reliably caught within the traditional test and debug process, something that has previously not been possible for software in general purpose computer systems. An entropy injector is built on a holistic debugger [Alb06b], a programmable debugging environment that provides non-intrusive access to all software state in a distributed system, and developers can use the holistic debugging services to create race condition stress tests (aka noise makers) guided by application-specific monitoring. Moreover, entropy injection places minimal requirements on the application under test, and is applicable for testing all types of applications, including distributed systems with heterogeneous hardware, operating systems, middleware, and programming languages, and for detecting errors involving multiple abstraction layers. We will mostly use simple shared variable accesses as examples; they are simple to explain and easy to reason about. The reasoning does not assume shared storage, however, and applies equally well to similar scenarios in all classes of systems.

We have built a proof-of-concept entropy injector, Njord, in the holistic debugging environment Nornir, which is based on the complete system simulator Simics [MCE+00]. We demonstrate Nornir with a unit test of a simple routine, and show how unguided random injection has little effect, whereas injection strategies that monitor application progress or targets a specific known bug provoke defects with high probability.

We will discuss race conditions in more detail and propose some new race

condition taxonomy in the next section. Nornir and holistic debugging will be described briefly in Section 3 and entropy injection is described in Section 4. Section 5 contains a description of Njord and the simple demonstration. Related work is described in Section 6, some discussion regarding entropy injection as a method in Section 7, and conclusions in Section 8.

2 Race conditions

There are a number of different definitions of race conditions (or *races* for short). In this paper, the most general definition the author was able to find will be used: "A condition in which two or more actions for an operation occur in an undefined order" [MBKQ96]. In other words, a race condition occurs in a concurrent system when two (or more) processes (A and B) race towards some points in their execution (Ac and Bc, respectively). No program construct controls whether Ac or Bc occurs first, and the execution path and the result of the program depends on the outcome of the race. Hence, a race condition makes the program execution and its results nondeterministic. Processes participating in a race need not be computational processes that execute instructions, but may also be other processes contributing to system progress, such as communication channels or clocks.

A race condition is born when some event in program execution happens and creates the race between A and B. We will call this event the *race opening*. It is usually the execution of an instruction in either A or B, but not necessarily. The opportunity of a race exists until either Ac or Bc occurs. We will call the event that either Ac or Bc happens the *race closure*, and the period of time between race opening and race closure a *race window*. Race windows are often short, and one closure is often much more likely than other closures. We will call this closure the *common race closure*. The common closure does usually not cause a program error — if it did, the developer could easily fix the error. We will call less likely closures *uncommon race closures*, or *erroneous race closure* if they cause an incorrect execution of the program. An erroneous race closure is also called *triggering a race condition*, since in many texts concerning race conditions, the uncommon closure is assumed to be erroneous. We will use the term *race event* as a common name for all events affecting race conditions. An execution of a concurrent system can be specified as the initial system state, the input to the system, and the interleaving of race events.

A race condition can involve more than two processes. In order to keep the examples simple, we will only describe races between two processes in this article. The reasoning and methods described in the article apply to any number of racing processes, however.

A common form of race condition is a *data race* [SBN+97, AHB03], which occurs when processes access a variable in shared memory without using a synchronisation mechanism that prevents simultaneous access, and at least one of the access operations is a write. In the example illustrated in Figure 1, two processes race to increment a shared variable X with a load, an increment, and

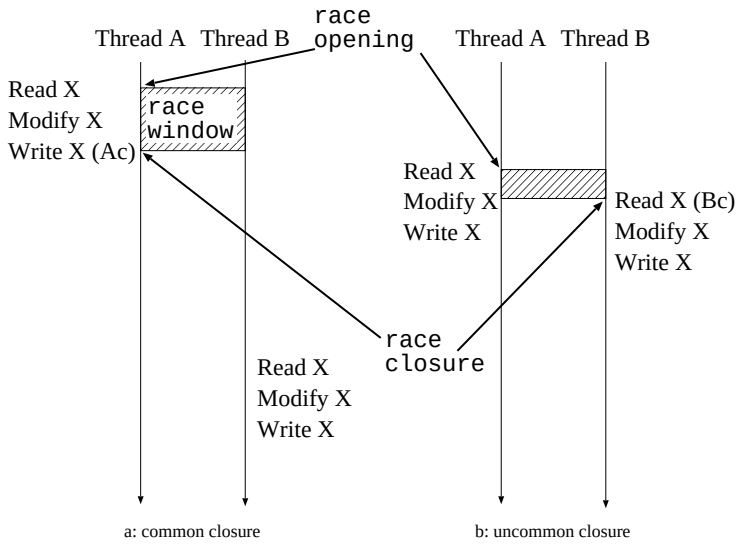

Figure 1: Data race.

a store instruction. The race window opens when a process executes the load instruction. The common closure occurs when the same process executes the store instruction. The uncommon closure occurs when the other process executes the load instruction (which opens another race as the processes race to the store instruction).

Some previous attempts to define race conditions are unnecessarily restricted to data races [NM92, HM94]. This is understandable, as data races is the only type of races that has received significant research attention. In this paper, we will instead use the broader definition above of racing processes connected in an arbitrary manner, which is applicable also to distributed systems without shared storage, and to race conditions that include non-computational processes.

2.1 Malign and benign races

A race condition is not a defect in itself. Most programs have expected nondeterministic behaviour. For example, in database applications, it is common to have multiple processes racing to commit transactions, and in multithreaded programs, threads often race to acquire locks. In these cases, all possible execution path resulting from a race condition are correct executions of the program. We will call these types of races *benign races*. In other cases, one of the execution paths leads to incorrect application behaviour. We will call such race conditions *malign races*.[1]

[1] Helmbold and McDowell [HM94] use the term *critical races*, which the author considers somewhat misleading.

There is no generally applicable method for distinguishing a malign from a benign race. Whether an execution path is correct or not is subjective. The most popular form of distinguishing correct execution paths from incorrect paths is to run test cases and check the result. The race detection method we present therefore assumes that the application developer supplies test cases and some mechanism for detecting incorrect executions. This requirement is typically easy to fulfill, since applications with quality requirements tend to have automated regression test suites, which can be used without modification.

There are classes of races that are uncommon in correct programs, and it makes sense to construct tools that report such races as potential programming errors. Tools for detecting unsynchronised accesses to shared data are discussed in the related work section 6.

Figure 2 illustrates pseudo code for a simple race condition that is neither obviously benign or malign. The output of the program will be "6" on most executions, but can also be "5", although it is unlikely. If "6" is the only expected output from this application, the race condition is malign, whereas it is benign if 5 is also an acceptable output. If we assume that the expected output is "6", and the race is malign, this example has two important properties: there is no concurrency pattern that can be identified as anomalous (the program uses correct synchronisation), and there is no result that can be easily identified as erroneous (classifying an incorrect execution requires an external oracle and a test harness). Many race conditions in industrial software have these two properties.

2.2 Classes of race conditions

Computer programs execute synchronously and deterministically most of the time. In order for a race condition to occur, there must be some nondeterminism, asynchrony, or concurrency present. Nondeterminism in computer systems can either come from the program itself, the underlying system, or from the external environment. We can classify causes for race conditions in the following groups:

Program concurrency Modern programs are often divided into multiple threads or processes on one or more physical machines. If these processes communicate, the program result becomes dependent on the scheduling between these processes, and on the arrival times of messages sent between them. The process communication can be explicit messages sent between the processes, or accesses to shared storage, such as main memory, files, database tables, tuple spaces, etc.

System nondeterminism There are various system services that have nondeterministic or timing-dependent behaviour. These fall into the following subcategories:

Asynchronous input/output Non-blocking and asynchronous I/O services depend on the availability of input data or free buffers, and produce different results depending on communication speed, which

```
lock L;
integer val;

a()
{
  do_small_amount_of_work();
  acquire(L);
  val += 1;
  release(L);
}

b()
{
  do_lots_of_work();
  acquire(L);
  val *= 2;
  release(L);
}

main()
{
  val = 2;
  t1 = thread_create(a);
  t2 = thread_create(b);
  join(t1); join(t2);
  print(val);
}
```

Figure 2: Example program with nondeterministic output.

is unpredictable. Services that poll for I/O completion, such as the `select` and `poll` system calls also fall into this category.

Interrupts Services that interrupt normal execution, such as hardware interrupts and Unix signals, may arrive at any point in execution, and therefore cause nondeterminism.

Unpredictable system services Some system interfaces have nondeterministic results by definition. Some examples are entropy-based random devices (e.g. `/dev/random` in Linux), clocks, and operating systems with randomised address layout.

Run-time system execution Underlying run-time systems, such as Java virtual machines, are designed to be mostly transparent, but in some cases, their execution can be observed by the software running in the virtual machine. For example, garbage collectors can affect applications at arbitrary points in execution, since they invalidate weak references and cause finaliser routines to execute. Moreover, virtual machines with dynamic optimisation can change the program code during execution if either the optimiser or the application has defects.

Asynchronous hardware While processors are mostly synchronous

from a software perspective, other hardware devices are not. Devices that perform direct memory access (DMA) can change memory contents at arbitrary points in execution. The memory system itself is asynchronous, and in a multiprocessor machine that provides a weak memory consistency model (which most high-end multiprocessors do), the cache coherence protocol may reorder memory operations nondeterministically.

Environment nondeterminism Modern computer programs are reactive and interact with the outside world, which is unpredictable in nature. Input may appear at any time and in any order, and if the program either receives interrupts when input arrives or polls for available input, the arrival time and order affects execution. Moreover, high-availability applications also need to handle faults in the services that the environment provides, for example hardware faults, timeouts, and communication errors. These may appear at any time during execution.

Although many of the examples mentioned above refer to low-level services, nondeterminism and asynchrony appear at all abstraction levels, and can generally not be avoided. Some asynchronous constructs, for example variables in shared memory, are results from design or implementation decisions. They can be circumvented with a different design, or by serialising the implementation at a loss of performance. In many application scenarios, however, the problem itself is concurrent and cannot be serialised. For example, an online store must be able to serve multiple customers simultaneously; it is not acceptable to serialise the implementation by having new customers wait until the first customer has decided what items to buy.

2.3 Regarding the likelihood of triggering a race condition

Time spent executing a race window is usually small in comparison to time spent outside race windows. If the race window is large, it can be triggered easily during testing, and developers can debug and correct the program with reasonable effort.

Figure 1a illustrates the typical execution scenario of a race condition — the race window opens by a process and closes shortly thereafter by the same process, and the likelihood that the race will be closed by another process and get triggered is very small. In the case of a typical data race, there is a read-modify-write sequence on each process, and the erroneous race closure only triggers if a read from another process takes effect between the read and the write. The race window is often small; in this data race example, the race window is only a few instructions long. In contrast, the number of instructions executed between the race windows is counted in billions or trillions. Hence, the likelihood of an erroneous race closure is very small. One might think that the probability of triggering an erroneous closure would be on the order of $10^{-}9$ to $10^{-}12$. Under some circumstances, however, the erroneous closure probability becomes high enough to occasionally trigger under execution.

As an example of a situation where the probability of an erroneous data race closure becomes higher than usual, assume that the code in Figure 1a runs on a large shared memory multiprocessor computer. If variable X is frequently read by most nodes in the system, the data caches of most processors are likely to contain the cache line corresponding to X in *shared state* in a MOSI cache coherence protocol, i.e. most processors have the permission to read variable X. When a processor enters the read-modify-write sequence, the load instruction is processed quickly, since the load memory operation is a cache hit. The store instruction, however, cannot be committed until the processor obtains the cache line in owned state in MOSI. If many other processors store the cache line in shared state, it may take a long time for the processor to receive write permission to the cache line, perhaps on the order of thousands of cycles. Hence, the probability of an erroneous race closure increases with several orders of magnitude.

Another example where erroneous closure probability increases is when the process executing the race window has to wait during window execution, for example due to an I/O request or a cache miss. As an extreme case, consider again the tiny race window in the read-modify-write sequence. If the sequence of instructions happens to cross a virtual memory page boundary in the machine, it may happen that the second memory page is absent from main memory and resides on disk. Since a disk request requires tens of milliseconds, the race window widens with about seven orders of magnitude.

Cache misses at various levels can also cause synchronisation among threads, which can increase erroneous closure probability. Consider the case where both processes in Figure 1 read the same file before executing the read-modify-write sequence. If the file is absent from the file system cache when the first process enters the sequence, it has to wait for it to be retrieved from disk. If the second process executes the file read operating during this wait period, it will also wait, and both processes will wake up when the file has been retrieved. Hence, the processes will execute the read-modify-write sequence at approximately the same time, which increases the probability of uncommon closure.

The bottom line of this section is that although erroneous race closures are very unlikely in the most common scenarios, there are plausible scenarios where they suddenly become likely and may affect software operation. These scenarios cannot easily be predicted, and preproduction software testing can therefore only catch a small fraction of the malign race conditions in an application. They instead appear in production scenarios, when the software experiences new load patterns, or is simply struck with bad luck, for example if race windows are placed on virtual memory page boundaries. Malign race conditions can be very expensive when experienced in production software, and there is therefore an incentive for spending resources on detecting them during testing, even though they are unlikely to appear in a given scenario.

3 Nornir

Nornir is a test and debug environment designed for all classes of applications, but focused on applications that have properties that make them difficult to debug with traditional tools: concurrent, distributed, reactive, real-time, and heterogeneous applications. Traditional quality assurance tools have fundamental limitations that prevent them from being applicable from these application classes, and from scaling to complex applications:

1. All factors affecting execution cannot be controlled, as discussed above. Therefore, intermittent defects often remain undetected in production software.

2. Test executions are not reproducible. Hence, even if an intermittent defect surfaces during testing, it cannot be reproduced and debugged.

3. The act of debugging and observing the system changes system behaviour. This is called *probe effect* [Gai85] and has limited the power and popularity of debuggers.

4. Traditional debugging tools only operate on a single process, at a single abstraction layer, whereas the application logic may be spread over multiple processes and abstraction layers, and written in multiple languages.

Item one and two are consequences of computer vendors' design decisions to build asynchronous and nondeterministic computers — exact instruction execution times, and in some cases the results as well, are not architecturally specified. Item three (and therefore also two) is a consequence of implementing observation facilities in the same system as the observed software. Item four is a side effect of items two and three. It does not make sense to build and use scalable debugging tools if the underlying observation services are unpredictable, and if increased usage of observation services causes degradation in the accuracy of observation due to probe effect.

Nornir therefore runs the software under test in a artificial, separate environment — an instruction set simulator, which is not affected by items 1–3 above. It is therefore a suitable base for building testing and debugging tools, and for creating scalable and automated debugging environments.

From a testing perspective, Nornir has a few unique properties:

- In Nornir, a failed test run can be reproduced and debugged with a variant of the GNU debugger that does not affect the execution. Developers can also write debugging routines that verify internal application state using holistic debugging services, described in Section 3.2.

- Since a simulator is implemented in software, the simulation model can be modified easily, and all factors affecting program execution can be controlled by the test engineers and regarded as test input to the application. In this paper, we show how testers can control ordering of all events, including those with undefined ordering. Similar techniques can be used for

controlling time flow, occurrence of faults, unusual responses from hardware or software services, etc.

- All software state can be observed from a programmable environment, as described in Section 3.2. This increases debugging efficiency, since developers can write debugging routines that validate internal state, and these can later be reused by other team members. It is also useful for writing white-box regression tests that compare internal state in different parts of a distributed application, and for steering test input based on the internal state of the application. Guided entropy injection, described in Section 4, is an example of such steering.

3.1 Instruction set simulation

A discrete event simulator makes simulation progress by processing events; in an instruction set simulator, the most common event is the execution of the next instruction. The simulator has a time model, and for each instruction processed, the simulated time is increased according to the time model. Time is quantified, and we call the smallest representable time unit a *simulation cycle* or simply a cycle. A cycle has a static time length, which often matches the period of a simulated processor, e.g. 10 ns for a 100 MHz processor. In this article, we will, unless otherwise noted, assume a simple time model, where each instruction requires a single cycle and memory operations execute instantaneously.

There are also other types of events that are processed and contribute to simulation progress. These are posted in event queues and processed when simulated time reaches the value corresponding to each event. Models for simulated non-processor entities use event queues to schedule the model processing for hardware devices, input and environment models, communication devices, and fault models.

3.1.1 Complete system simulation

Instruction set simulators have suitable characteristics to serve as debugging platforms: they provide models that are detailed enough for running applications in binary form, but still run sufficiently fast to run large applications. We have built Nornir on Simics [MCE+00], a complete system simulator (also called full system simulator). A complete system simulator has models for all hardware devices that provide services visible to software, i.e. processor, memory, disks, network cards, etc. It is binary compatible with general purpose computers, and runs unmodified commodity operating systems and applications in binary form.

Simics can simulate multiprocessor machines and multiple networked computers, which may be heterogeneous in architecture. It is designed to be deterministic; if a simulation scenario starts in a well defined simulation state, and the input model is synthetic and predictable, it will produce exactly the same scenario for each simulation.

Simics runs roughly two orders of magnitude slower than the host machine when modelling a machine similar to the host. It is slow enough to be a significant drawback for simulation as a method, but fast enough to allow large applications to be observed. The simulation speed depends on the accuracy of the timing model. We usually run with a coarse model, where every instruction takes one clock cycle. This model is sufficient for detecting logical errors, which is the scope of this article. If the simulator should be used for performance analysis, a more accurate timing model may be desirable. Simics allows users to model cache memory hierarchies, providing a good timing approximation without sacrificing much simulation performance. It also supports detailed models of processor pipelines, out-of-order execution, and speculative execution, at the price of severe performance degradation. Magnus Ekman's dissertation [Ekm04] contains some benchmarks on Simics with different timing models. Complete system simulator timing models have been validated and discussed by Gibson et.al. [GKO+00].

Simics has two performance features that reduce the cost of simulation in many scenarios. First, the entire state of the simulated system can be stored to disk in a checkpoint and later retrieved. This feature may cut down execution time in scenarios where the test setup is costly, but may be reused for multiple tests. Simics also performs idle loop detection, and fast forwards processors waiting in the operating system's idle loop to the next asynchronous event. This improves simulation performance for I/O-bound workloads and many distributed application scenarios.

A complete system simulator allows users to control all types of nondeterministic behaviour mentioned in Section 2.2, either through services exported by the simulator, or by developing simulation model extensions. The Simics configuration used for the demonstrations below uses standard Simics device and clock models, and a sequentially consistent memory model.

3.1.2 Time and clocks

If the simulated system has multiple processors, the simulator executes instructions on the processors in a round-robin order. Hence, the resulting simulation is a serialisation of the concurrent execution of a distributed application. In a simple simulator implementation, there will be a single simulated clock, and each processor will execute at most one instruction in each simulation cycle. If the processors have different frequencies, or if some instructions take more than a cycle, some processors will stall in some of the cycles. Simulating all processors each simulation cycle, however, has negative implications on simulation performance, as it prevents some efficient implementation techniques. Therefore, in an efficient simulator implementation, the simulator will execute a large number of instructions on one processor before switching to another processor. This simulated time period is called a *time quantum*, and is often on the order of a thousand cycles or more. In such an implementation, each processor has its own clock, and the clocks temporarily drift apart, at most a time quantum.

If the time quantum is small, software will not notice that there is a slight

drift between the processors. If the time quantum is large, on the order of seconds, and if multiple simulated clocks can be observed by software, it is possible to write programs that observe time going backwards. In such scenarios, the simulator may induce incorrect software behaviour, but these situations are fortunately rare.

Software running outside the simulated world, such as device models and observation software, can also observe multiple simulated clocks, which are temporarily unsynchronised. Hence, when writing such software, one needs to be careful and ensure that the software can handle decreasing time values. When developing Nornir, we have several times noted that it is easy to make a mistake and assume that clock readings always provide increasing values. In order to simplify implementation of observation software, we have introduced two artificial clocks: *head clock* and *tail clock*, and the corresponding time scales *head time* and *tail time*. The head clock value is the largest value of all simulated clocks, and the tail clock value is the smallest simulated clock value. These artificial clocks are monotonically increasing, and are therefore simpler to use than dealing with multiple unsynchronised clocks tied to processors.

3.1.3 Memory consistency

A simulated symmetric multiprocessor machine with a shared main memory implements a sequential memory consistency by default.[2] The processors directly access a single image of main memory, and the interleaving of memory transactions follows the interleaving of simulated processor execution. If the simulated machine includes a model of a cache memory hierarchy, the cache model typically only affects the time model, and does not model consistency effects of data caching; the memory transactions still go directly to main memory, but the instruction requires more than a single cycle.

A simulated machine with a sequential consistency model is insufficient if a developer wants to test whether a shared memory application works correctly in the presence of memory transaction reorderings on a multiprocessor machine with a weak memory consistency. In order to test for software defects resulting from memory transaction reordering, one needs to implement a memory consistency model where processors observe different memory images. Implementing such a memory consistency model is possible if the simulator allows user extensions to intercept and modify memory transactions. Such a model could be created by improving upon an existing time model that implements a realistic cache coherency protocol model, and make it also serve processors with cached data. An alternative approach would be to add reorder buffers to and/or from processors and any shared caches. These buffers could stall and reorder data according to the architectural memory consistency model for the

[2] Actually, simulated multiprocessor machines provide an even stronger consistency model, linearisability (with respect to a monotonically increasing global time, i.e. head time, tail time, or time on a single processor). Software in the simulated system cannot distinguish between sequential consistency and linearisability, however, and it is hard to come up with a simulated scenario where it makes a difference, so the issue is mostly of academic interest.

simulated machine. This could result in a complex implementation in the general case, for relaxed consistency models. Stronger consistency models, such as total store order (TSO) could probably be modelled with few and simple reorder buffers, however. In the case of relaxed memory consistency models, it may be more straightforward to use a separate copy of the entire main memory for each processor, and have reorder buffers that stall modifications from other processors before they modify each processor's main memory. Simics does provide the necessary services to intercept and modify memory transactions, and also a memory consistency controller, but the author is not aware of any instruction-set simulator implementations of weak memory consistency models for the purpose of software testing, and this area is future work.

3.2 Holistic debugging

Nornir is a framework designed for programmable, non-intrusive observation of all system state that is visible to software, even in complex and distributed systems. We call such an observation environment a *holistic debugger* [Alb06b]. A holistic debugger does not assume any properties of the system under test, except that it must be feasible to simulate the hardware and external environment (e.g. interactive users, sensor input). It assumes nothing about the software, and supports non-intrusive debugging and comparisons of execution state in multiple programs, written in different languages, at multiple abstraction levels, running in multiple, heterogeneous computers, etc. Our implementation is of course limited to a few configurations for practical reasons, but the conceptual method is free from the inherent limitations of traditional debugging and analysis methods.

A complete system simulator provides non-intrusive access to all system state visible to software. Unlike standard debuggers, which use probing services supplied by the runtime system to probe the state of running processes, the holistic debugger must use non-intrusive probing techniques, and cannot rely on runtime system services. It probes the simulator for machine state, but the information retrieved is raw, binary information. The information has been obfuscated by symbolic transforms, i.e. compilers, and by machine transforms, i.e. virtual machines and operating systems, and it is no longer easily comprehensible. In order to make this information useful for a programmer, it must be translated back to the abstraction level the programmer deals with, i.e. to variables and types in the programming languages used in the application.

In a holistic debugger, there is an associated *abstraction translation stack* for each inspected process in the simulated system. A translation stack consists of symbol context probes and machine context probes, corresponding to the language environments and virtual machines of the inspected process. The structure is shown in Figure 3. When the user inspects a particular program, a translation stack is instantiated. The top of the stack is a symbolic probe that lets the user inspect the execution and state of the program, similarly to a standard debugger. The symbolic probe queries the underlying machine context probe for program state data. If the machine context refers to a physical

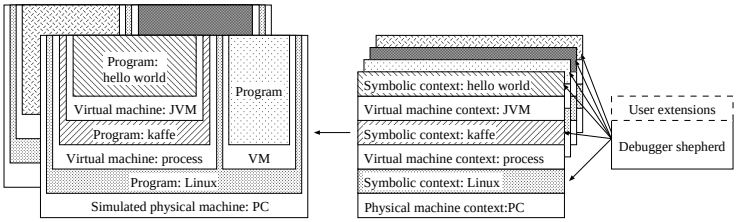

Figure 3: Holistic debugger structure with example applications.

machine, the machine context probe forwards the query to the simulator. If the machine context below is a virtual machine (e.g. a Java Virtual Machine or a Unix process), however, the corresponding machine context probe is a *virtual machine translator* (VMT). The VMT parses the virtual machine's data structures, and maps virtual machine storage to storage in the underlying machine. This parsing is done through the symbolic context probe for the underlying program providing the virtual machine.

Nornir supports debugging applications in simulated Sparc/Linux systems, written in a language supported by GDB. GDB acts as symbolic probe for the applications. GDB requires a backend service that allows it to control and probe a single Linux process. It therefore communicates with a Linux VMT that translates GDB's requests for virtual memory and register contents to the corresponding physical memory and registers contents, which are retrieved from Simics. Breakpoints and other basic debugger services are handled similarly. The VMT also exports services for suspending execution on events that are not tied to a specific process, for example the start of a particular application or arrival at a certain point in time. We call such generic breakpoints *eventpoints*. When probing the kernel data structures, the VMT uses a *static symbolic translator*, a set of automatically generated classes that contain methods for parsing kernel data structures in binary form. There is one such class for each type in the kernel, and the static symbolic translator thereby provides a robust, strongly typed programmable probing system for the kernel.

Abstraction translation stacks enable users to probe software state at arbitrary level, in any process, without affecting the execution. In order to debug a distributed application, the user can attach a symbolic debugger to each participating process, or write automated debugging routines. In a holistic debugger, all symbolic probes are connected to a debugger shepherd, a programmable global debugger that allows debugging routines to probe the state of the entire system. Nornir's debugger shepherd, *Verdandi*, is written in Python and provides a Python programming interface. It creates an instance of GDB for each process the user wishes to monitor. The user validates internal state by writing monitoring routines, called *causal path monitors*, that observe a data flow path by waiting for a set of eventpoints to trigger, probe application state via GDB, wait for a new set of eventpoints, etc. Causal path monitors run independently of each other and concurrently. The causal path monitors can be saved an reused

in future debugging sessions, or used as white-box tests in regression testing. The user can also define application-specific eventpoints, and use causal path monitors to determine when the eventpoints should be triggered. Examples of useful application-specific eventpoints would be the arrival of a specific class of http request or an SQL query matching a specific pattern.

A holistic debugger is a program observation framework that places minimal requirements on the system under test, and it works for arbitrary distributed software systems. Nornir is a research prototype and has limited capabilities that are suited for our experiments, but the method itself makes very few assumptions. It assumes that it is feasible to simulate the system under test and its external input, and the holistic debugging services require a virtual machine translator for each abstraction layer involved. In practice, this requires source code access to the virtual machine, or that the virtual machine or operating system vendor supplies a translator. Source code access to all parts of the application is not required, however, and errors in binary components can be detected, reproduced, and reported to the vendor.

The holistic debugging concept and Nornir's implementation is described in more detail in the author's previous works [Alb06b, Alb06a].

4 Entropy injection

In order to detect race conditions during testing, one must put the software under more stress that it will experience when put into production. Trying to increase the general stress level by raising the load is not effective — it is likely that the program would execute a few set of execution paths and event interleaving patterns over and over again. Instead, one should strive to push the software under test to go through new event interleavings that can possibly appear when the software is put in production, but are unlikely to appear during lab testing. This is technique bears some resemblance to *fault injection*, a well established technique for testing high-availability features. Instead of injecting component faults that the application is supposed to handle, however, one injects timing chaos, and ensures that the application handles it gracefully. We will therefore call this technique *entropy injection*, and the mechanism or tool that performs the injection will be called an *entropy injector*.

Consider again the common data race scenario in Figure 1a. The race window is small and the program is spending most of the time executing outside the race window. Hence, the uncommon closure is unlikely to occur. In order to determine whether the application executes correctly in both possible scenarios, one should provoke the uncommon race closure, run the application to completion, and have a test oracle determine the correctness. By preventing thread A from making progress, the race window is artificially expanded and the uncommon closure is provoked, as illustrated in Figure 4. In the general case, controlling computation progress is difficult or infeasible in real computers, but with a computer system simulator, one can modify the machine model and achieve an artificial delay in the process executing the race window. How to

perform this injection in practice is described in Section 4.1.

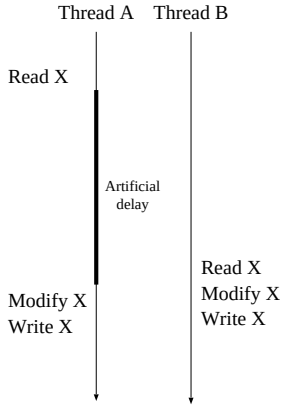

Figure 4: Provoking a race condition by artificial delay injection.

If the application is run in a simulator, and the location of race windows are known, one can provoke different race closures in an application, and use the test suite's oracle to determine whether the application is correct or not. Unfortunately, developers do not know where the race windows are; if they did, it would be easy for them to correct the program. Instead, in order to test for race conditions, the locations of race windows must be guessed by the injection mechanism, potentially guided by an application developer.

Searching through the whole state space of all possible interleavings of race events is usually infeasible; the state space is too large for applications with non-trivial communication patterns. Instead, the injector inserts delays at points in execution that could be within a race window. It should avoid repeatedly testing the same potential race windows and the same interleaving patterns, and strive towards test coverage in terms of event interleaving. For example, if one wants to test for malign races resulting from interleaving of communication in a distributed application, inserting multiple delays in one process only improves coverage if the process has communicated with other processes between the delays. Hence, the injector needs a strategy in order to make educated guesses on where to insert delays that provoke application errors. Entropy injection strategies are discussed in Section 4.2.

An injector inserts delays randomly, with a pseudo-random number generator. When an error has been provoked and reported by the test suite, the developer can reproduce the failing execution by using the same random seed, and debug the error in the traditional cyclic manner, either interactively with a non-intrusive symbolic debugger, or programmatically with a holistic debugger.

Since entropy injection relies on the application's test suite to distinguish correct executions from incorrect executions, there will be no false positive reports. Thus, even in cases where there is an apparent race condition, for example a data race where a variable is accessed without proper synchronisation, but the

application contains logic for preventing unsynchronised accesses, or if all race closures result in an execution that is deemed correct, the test suite only looks at the final result and does not produce false error reports. The absence of false positives is an important improvement over static and dynamic race condition detection tools (described in Section 6, which can only warn about suspicious concurrent program behaviour according to heuristics, and tend to produce numerous false positive reports.

4.1 Execution flows

In order for an injector to affect the interleaving of unsynchronised events, it can either modify the state of the software, or control the progress of simulated entities in the system. Modifying the software state is undesirable for a number of reasons: It means that the released software is not the software tested, it can complicate reasoning about test coverage, and it can make it difficult to determine whether provoked errors can appear in real systems. Since all interleavings of race events that are feasible in a real machine can be provoked by controlling the progress of simulated entities in an appropriate manner, the latter method is sufficient for most test scenarios, and the one we have pursued.

The simulation model can be thought of as having multiple *simulation flows*, where each flow corresponds to the progress of a simulation entity. If a flow is enabled, the modelled entity performs as usual, but if the flow is disabled, the entity is stalled and makes no progress. In normal simulation, all flows are enabled and the simulation progresses as usual. An injector may disable flows in order to provoke new interleavings of machine events. The progress of a flow can affect the execution actively, for example by inserting event notifications in the event queue. It can also affect the execution passively, e.g. a modelled temperature sensor that only yields values when polled by software. For a passive flow, progress means that subsequent polls may yield different values; a disabled passive flow yields a fixed value. Simulation flows can be categorised as follows:

Processor flows There is a processor flow associated with each simulated processor. Processor flows only affect the simulation actively, by executing instructions. A disabled processor flow corresponds to a stalled processor, which is prevented from executing instructions.

Device flows Each simulated device processes requests from software and responds through interrupts, direct memory access (DMA), or by memory-mapped I/O. A device flow can be both active and passive. A disabled device flow inserts no events in the event queue, and values read from its mapped memory region remain fixed.

Input flows The simulation scenario has a synthetic input model for each of the external sources of input affecting the simulation, for example human users, network sources, and temperature. Since external input is received

by the simulated computers by hardware devices, input flows overlap with device flows.

Communication flows Each communication link between nodes in the simulated system has its corresponding flow. When a communication flow is disabled, the data on the link are queued and delivered when the flow is enabled again.

Clock flows The flow of time also affects simulation, and race conditions where a process races against the clock are common. Hence, a complete entropy injector should be able to achieve arbitrary interleavings with the progress of processes and the progress of simulated time. In order to achieve arbitrary interleavings, the simulator must be able to halt a clock while executing instructions on the corresponding processor, and to forward the clock without executing instructions.

Memory consistency flows A memory consistency flow corresponds to data transfers in the cache coherency mechanism of a multiprocessor machine. Disabling and enabling memory consistency flows affects the order in which memory operations are observed.

Fault injection flows Each simulated device with a fault injection model has a corresponding fault injection flow. Faults are only injected if the flow is enabled.

A particular class of flows can be enabled and disabled if the simulation model provides an appropriate service or allows supplying user-defined models for entities. Adding the ability to control clock flows and memory consistency flows require some support from the simulator core for an efficient implementation, but controllability for the other classes of flows are easy to implement.

At each point during simulation, a non-empty set of flows are enabled. We will call such a set of enabled flows a *flow configuration*. An entropy injector alternates between flow configurations, and thereby achieves desired interleavings of hardware events. It is not necessary that all possible flows in a simulated scenario are considered in the flow configuration selection process. It may be the case that some flows do not significantly affect the software under test, or that the simulator implementation does not allow control over some flows. Flows that are not explicitly controlled by the injector are either implicitly enabled or connected to another flow.

In most test scenarios, detecting the race conditions that can appear would only require controlling one or a few types of flows. When testing multithreaded applications on simulated multiprocessor computers, controlling the processor flows is a simple strategy for triggering races between multiple threads. Individual threads can be suspended by disabling the processor they are currently running on. If a multithreaded application runs on a simulated uniprocessor computer, however, disabling the only processor does not achieve the desired effect. Instead, the injector must alternate disabling the processor flow and the

clock flow in order to provoke the operating system to schedule a new thread when desired. In a test scenario with an application distributed over multiple nodes, controlling the time flow is less important for provoking race conditions, and controlling processor flows, communication flows, and communication fault injection flows may be more appropriate. Some classes of flows, such as memory consistency flows and hardware fault injection flows, will usually not be necessary to control, except for certain classes of applications.

The flow model matches the simulation setup well when the simulation has a static set of flows that correspond to simulated entities that all process events serially, e.g. processors or input devices. It does not match equally well to entities that do not process events serially, e.g. communication links that reorder messages, memory systems with weak consistency, etc. One could support a dynamic number of flows, or map dynamic flows to a static set of flows, but it could complicate the injector implementation and potentially have performance implications. The author believes that the flow model will have to be revised and extended as more experience is gained with entropy injection, but that it is sufficient for creating entropy injectors that are theoretically capable of catching most classes of race conditions.

The flow classes mentioned above are all low-level flows, corresponding to the level of simulation that we are using. If a simulator that model computers at another abstraction levels is used, flows at that abstraction level would be appropriate. For the purpose of testing application race event interleavings, it may be more useful to control flows at higher abstraction layers, for example processes or database transactions. This is discussed further in Section 4.2 and there are examples in Section 5.1.

4.2 Injection strategies

An injector can insert artificial delays in execution by alternating between flow configurations. Inserting delays completely randomly is unlikely to be effective in most cases, however. If an application test suite with an injector is run infinitely many times, all intermittent defects will be found, if the test suite can detect them and the injector can provoke them with the set flows it controls. Unfortunately, since the number of possible race event interleavings is large, it would take an unreasonable number of test executions to find all race conditions in an application.

An entropy injector therefore needs an *injection strategy* that searches through the space of possible interleavings in an intelligent manner, inserts delays where the application is likely to be vulnerable to race conditions, and avoids excessive testing of redundant test cases that have equivalent interleavings of application events. A strategy is comprised of multiple combined *injection tactics*. Most tactics are *provoking* and have notions of where race windows are located, and strive to insert delays in randomly picked race windows. If a race window is hit and a sufficiently large delay is inserted when the window is open, there is a good chance that some process will execute an uncommon race closure. For example, a tactic for provoking concurrency errors in a database

could insert delays in processes that have started transaction processing, but not yet completed commit phase. A provoking tactic typically concentrates on a particular type of event that is assumed to be important for race event interleavings in the application under test, and distributes injections with uniform probability over these events.

In theory, any instruction or event in a process in the system can affect other processes. A very simple injection tactic could have each flow enabled for a random number of instructions (or other events), and then disable the flow until a random number of instructions have been executed in other flows. We will call this tactic *RandomTime*. If the random numbers are picked from an open-ended random distribution, e.g. an exponential distribution, RandomTime can trigger any intermittent error. RandomTime regards all executed instructions as equally important, and inserts evenly distributed delays over the dynamic stream of executed instructions. RandomTime is a simple but *unguided* tactic, and therefore not sufficient for triggering defects on its own, which is demonstrated in Section 5.2.

In practice, only a few types of events will affect interprocess communication in a given application. These events are typically unevenly distributed over the instruction stream, and RandomTime will therefore produce many equivalent and redundant test cases while missing relevant test cases. As an example, assume that a developer wants to test a distributed application for defects that depend on the order of messages sent between the nodes. RandomTime would probably generate many redundant test cases where delays are injected at different points in execution that have no communication event between them, illustrated in Figure 5b. It would also miss injecting delays in many interesting cases where there are two communication events separated by a small number of instructions. Such cases, shown in Figure 5c, are desirable to test and increase test coverage.

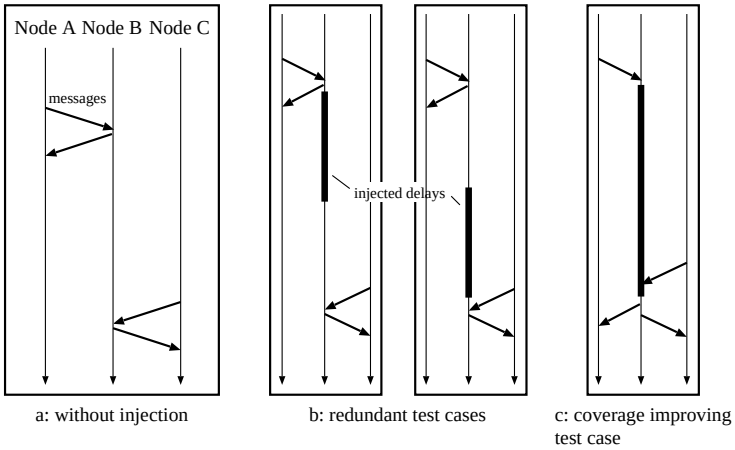

Figure 5: Redundant and coverage improving test cases.

An injector will be more effective if its strategy involves *guided* tactics, which monitor execution events that are likely to affect race event interleavings, and distribute delays randomly and evenly over these events. Such tactics can be general, and insert delays at events that are likely to be race events in many applications, e.g. shared variable accesses or interprocess communication. They can also be domain- or application-specific, and insert delays at application events known to be sensitive to races, for example at telephone call state transitions, during online software upgrades, or during database transaction commit phases.

A tactic that only needs to monitor events at the hardware level can be implemented on a simulator without a holistic debugger. Tactics for provoking defects in high-level applications, however, will need to monitor events on an abstraction layer above the hardware, such as events in a Unix or Java process, and must rely on a holistic debugger to provide the necessary programmable debugging services. A tactic can follow the progress of an execution flow at a higher abstraction level by using holistic debugger services to place process-specific eventpoints on the monitored process, e.g. code breakpoints, data watchpoints, or time breakpoints. When the tactic determines that the monitored process should be delayed, it manipulates an appropriate hardware flow to achieve this effect. It is straightforward to delay a process by disabling the processor it is running on. It is also possible to affect operating system's process scheduling by fast forwarding the clock, thereby forcing timer interrupts and thread preemption. Similar techniques may be used in order to force other run-time systems to control flows at higher abstraction layers.

If a holistic debugger is unavailable, it is possible to create other means for monitoring high level application events and controlling high level flows, by manipulating the code or the state of the operating system or the application under test. Such test scenarios would be intrusive, however, and the software under test would differ from production software. Intrusion has a number of disadvantages; it may mask defects that cannot be found due to the modifications, it may trigger defects that cannot appear in real systems, it complicates test configuration, and it complicates reasoning about test coverage. In some cases, however, it can be a more cost-effective solution if the holistic debugger can be replaced with simpler debugging services.

An aggressive injector can create simulated scenarios that can never appear on real machines, and these scenarios can potentially trigger false defects that can never appear in production. In such scenarios, some device exhibits behaviour that violates the assumptions made by the application designers. For example, if an injector delays a disk device and it therefore takes a few seconds for it to respond, the operating system device driver may report a timeout error, and if the application does not handle the error gracefully, it will fail. In this case, if the application designers has decided that disks are assumed to work, and that any failures resulting from disk failures are acceptable, this test case is invalid, and should not be reported as a failure. Applications can also fail if the simulated machine violates some basic assumptions made by software. Consider a simulated scenario with multiple processors, each having its own clock.

If the simulated clocks are artificially delayed and drift apart, the operating system could perform a clock reading on one processor, and later another clock reading on another processor, this time resulting in an earlier time value. Such anomalies may confuse the operating system or other software on the system, and cause application failures.

In order to avoid creating false positive reports, developers can write detection routines that identify invalid test cases when a timeout or similar undesired execution pattern appears, and discard the results of these tests. An alternative method for avoiding false positive reports is to use *restrictive tactics*, which prevent the simulated machines from exhibiting unrealistic behaviour by limiting the duration of artificial delays. Such a tactic would monitor the progress of a simulated entity, and if the entity is delayed for an unreasonable amount of time, or falls behind related entities, the restrictive tactic would enforce the entity to become enabled in order to avoid anomalous system behaviour.

An injector holds a number of active tactics, combined to a strategy. The tactics of a strategy have different priority, and high priority tactics get the first opportunity to affect the flow configuration. Lower priority tactics can decide to enable or disable flows that have not been decided by higher priority flows. Restrictive tactics need to have high priority in order to take effect. The guided tactics that are expected to accurately guess the likely locations of race conditions should have medium priority. Unguided tactics can be added at a low priority, with the purpose to add noise and provoke local variations of similar execution patterns.

Tactics can also be composite, and combine multiple subtactics. A composite tactic could use all subtactics or alternate between them, either randomly or depending on application behaviour. Composite tactics allow developers to create arbitrary hierarchies with tactics.

5 Njord

We have implemented a simple entropy injector, Njord, as an extension to Nornir. The purpose of Njord is to demonstrate a proof-of-concept implementation, and to gain some understanding of how the method behaves in practice by studying test executions with simple, synthetic test scenarios. Njord injects artificial delays by controlling processor flows, which is sufficient for provoking most race conditions on multiprocessor computers and in distributed systems. Other flows in the simulated system, e.g. clocks, hardware devices, etc, are enabled and progress as usual according to the simulation model. Njord supports the same simulated platform that Nornir supports, i.e. simulated UltraSPARC systems running Linux.

5.1 Njord injection strategies

Njord implements two simple, generic entropy injection tactics. More complicated, application-specific tactics are likely to be more effective for realistic

applications, but these simple tactics allow us to demonstrate how an application developer could provoke race conditions and write a regression test case, and to gain some understanding of entropy injection behaviour.

A Njord injection strategy consists of a list of tactics, ordered by priority. Tactics are responsible for monitoring execution events they consider interesting by using appropriate eventpoints, and can decide to request flow configuration reevaluation when an eventpoint is triggered. When reevaluation is requested, Njord iterates over the tactics in priority order. Each tactic can set a flow to enabled, disabled, or leave it to be decided by lower priority tactics. No tactic may disable all flows. After consulting all tactics, Njord sets all undecided flows to enabled, imposes the new flow configuration on the simulator, and resumes simulation.

Tactic implementations use holistic debugger services to monitor events of interest. Since a holistic debugger can monitor arbitrary software events, tactics can react on any type of eventpoint in a simulated system, e.g. code execution, memory access, time eventpoints, process creation and termination, file access, network message reception, web server query, and database table access. Hence, injection strategies can be used for provoking malign race conditions at any abstraction level.

5.1.1 RandomTime

The first implemented tactic is RandomTime, as described in Section 4.2. Pseudo code for RandomTime is shown in Figure 6. It imposes a flow configuration with a random set of processors enabled, waits for a (exponentially distributed) random number of cycles, and repeats the procedure. RandomTime has two configurable parameters: the number of processors enables (ProcessorsEnabled) and the average delay period (Delay).

```
while (true) {
  enableAllProcessors();
  while (numProcessorsEnabled() > ProcessorsEnabled)
    disableProcessor(random() \% numProcessors());
  do {
    nextFlowChange = currentCycle() + randomExponential(Delay);
    awaitOneOf(CycleCount(nextFlowChange),
               FlowConfigurationChanged());
  } while (currentCycle() != nextFlowChange);
}
```

Figure 6: Pseudo code for RandomTime tactic.

As discussed below, RandomTime alone is not sufficient for provoking defects, but it is useful for adding random noise to the entropy injection, which can prevent all test runs from reexecuting a small set of patterns. It can also be used as a restrictive tactic in order to limit the delays introduced by other tactics, in this case with a long Delay setting and ProcessorsEnabled set to the number of processors in the system.

5.1.2 RandomInstruction

The second implemented tactic, RandomInstruction, spreads delays evenly in a static set of instructions, in contrast to RandomTime, which spreads its delays evenly over the dynamic set of executed instructions. The assumption of RandomInstruction is that there is a race window open when a certain code statement is executed, and if the injector injects a delay when that statement is executed, the race is likely to manifest. The pseudo code for RandomInstruction is shown in Figure 7. The basic algorithm is simple: Set a number of breakpoints at random addresses in the application's processes. When one of the breakpoints are hit, inject a delay in the corresponding process by disabling the processor that hit the breakpoint. Disable all breakpoints, wait for a random number of cycles, enable the processor again and reenable the breakpoints that have not yet been not hit. The breakpoint addresses are automatically obtained from debug information generated when the program was compiled.

RandomInstruction has a number of parameters. The AddressRegions parameter allows developers to specify the set of instructions that are candidates for breakpoints. The current implementation of RandomInstruction is adapted to multiple threads that run identical program code in one address space. An implementation for processes with multiple address spaces would be slightly different, but have similar complexity. The Fraction parameter specifies the number of breakpoints, as a fraction of the address region size. There is also a Fallthrough parameter, which the developer can use to specify a probability that a breakpoint should be ignored when hit. This gives some chance of triggering defects that do not appear the first time the faulty piece of code is executed.

When performing a full application test, the normal procedure would be to run the application with an injector, and then repeat the whole scenario a number of times, but with different random seeds. When testing multithreaded functions or modules, however, calling the module under test multiple times from a test driver program is more efficient. In this case, the injector should reconsider the set of breakpoints between the invocations of the module under test. There is therefore a ProcessThreshold parameter, and whenever the number of application processes drops below this number, the tactic restarts the algorithm, ensuring that each iteration of the module test is run with a new set of breakpoints. The ProcessThreshold parameter is not necessary for RandomInstruction to operate, but illustrates how developers can use application knowledge to improve test performance.

If processes in the application under test communicates with a regular mechanism, for example with global variable accesses, network messages, or database accesses, developers can create similar strategies for spreading injections evenly over communication events deemed important. In this case, the strategy would follow the same pattern, but the execution breakpoints would be substituted with variable read/write watchpoints, message eventpoints, or database access eventpoints, respectively.

```
while (true) {
  if (numApplicationProcesses() == 0)
    awaitOneOf(ProgramStart());
  else if (numApplicationProcesses() < ProcessThreshold)
    awaitOneOf(ProcessCreation(), ProcessDestruction());
  else {
    for (i = 0; i < max(AddressRegions.size() * Fraction, 1); ++i)
      breakpoints.add(insertRandomBreakpoint(AddressRegions));

    while(numApplicationProcesses() >= ProcessThreshold) {
      event = awaitOneOf(BreakpointHit(breakpoints),
        ProcessDestruction());
      if (event.type() == BreakpointHit) {
        if ((random() \% 1000) > (Fallthrough * 1000)) {
          breakpoints.remove(event.breakpoint());
          disableProcessor(event.processor());
          delayEnd = cycleCount() + randomExponential(Delay);
          while (currentCycle() < delayEnd) {
            delayEvent = awaitOneOf(CycleCount(delayEnd),
              ProcessDestruction());
            enableAllProcessors();
            if (delayEvent.type() == ProcessDestruction) {
              if (numApplicationProcesses() < ProcessThreshold()) {
                breakpoints.clear();
                break; // Ignore delay and wait for process creation.
              }
            }
          }
        }
      }
    }
  }
}
```

Figure 7: Pseudo code for RandomInstruction tactic.

5.2 Demonstration

In order to demonstrate how entropy injection behaves in practice, we have arranged a scenario with a unit test for a simple multithreaded routine that contains a race condition. The routine, `increment_n`, creates `n` threads that perform some random work, and then increment a shared variable, `sum`. The variable increment is performed without using synchronisation, and there is a typical data race, as described in Section 2. In case the uncommon window closure occurs, it is detected by comparing `sum` with `n`. Figure 8 shows the pseudo code for `increment_n`, the routines it calls, and a unit test. The numbers S and R change for the experiments.

The example is contrived, but is representative in the sense that the race window is very short, the program performs lots of activities in between, and the race only triggers if the threads arrive simultaneously, which is unlikely but plausible.

```
increment_n(int N) {
  for (i = 0; i < N - 1; ++i)
    threads[i] = thread_create(increment_one);
  increment_one();
  for (i = 0; i < N - 1; ++i)
    thread_join(threads[i]);
}

increment_one() {
  spinAmount = random() \% S;
  for (i = 0; i < spinAmount; ++i)
    doRandomWork();
  ++sum; // Lock missing here.
}

doRandomWork() {
  switch (random() \% x) {
    case 0: readAFile(); break;
    case 1: writeAFile(); break;
    case 2: doSomeSystemCalls(); break;
    case 3: fibonacci(); break;
    case 4: useSomeMemory(); break;
    /// And so on.
  }
}

test_increment_n() {
  E = 0;
  for (i = 0; i < R; ++i) {
    sum = 0;
    increment_n(2);
    if (sum != 2)
      ++E;
  }
}
```

Figure 8: Pseudo code for the test example.

5.2.1 Test results on real hosts

When run on real machines, the race in the example above triggers, but only rarely. Statistics from running the test program on a variety of multiprocessor machines, ranging from very old to modern, are shown in Figure 9. The following machines were used for the test runs:

brahma Sun SPARCstation 10, 2 x 40 MHz SuperSPARC

scheutz Sun Ultra Enterprise 4000, 8 x 250 MHz UltraSPARC II

r2d2 2 x 600 MHz Intel Pentium III

millennium Sun Fire T1000, 1 x 1 GHz UltraSPARC T1, 6 cores, 4 threads/core

host	S	E	seconds
brahma	1	0	253
brahma	10	0	3057
brahma	100	0	42K
brahma	1K	2	126K
scheutz	1	0	57
scheutz	10	0	691
scheutz	100	0	19K
scheutz	1K	0	181K
r2d2	1	21K	162
r2d2	10	0	644
r2d2	100	0	2885
r2d2	1K	1	81K
millennium	1	0	117
millennium	10	0	1573
millennium	100	0	14K
millennium	1K	0	117K
saruman	1	0	5421
saruman	10	0	3270
saruman	100	0	37K
saruman	1K	0	59K
scaramanga	1	0	20.1
scaramanga	10	0	141
scaramanga	100	0	1145
scaramanga	1K	0	20K

Figure 9: Race statistics, real machines. R = 1M for all runs.

saruman 1 x 2.4 GHz Intel Pentium 4, 2 threads/core

scaramanga 2 x 3.0 GHz Intel Xeon, 2 threads/core

In most of the configurations, the race never triggered. In some cases, the speed of thread creation happens to match the amount of work done by the main thread, and there is a realistic chance of an uncommon race closure. As we can see in the section below, even simple injectors improve the chance of erroneous race closure significantly. The figures presented here should not be interpreted as a quantitative comparison between testing on real and simulated machines; debugging race conditions with standard methods is not an option, since errors cannot be reproduced.

5.2.2 Test results with entropy injection

We have run the example test routine above in Nornir, using injection strategies with different variations of RandomTime and RandomInstruction tactics. The simulated machine, `bagle`, is a dual processor 167 MHz UltraSPARC II running SuSE Linux 7.3, obtained from a standard Simics 3.0.23 distribution. Five different injection strategies have been used, with two different values for the `Delay` parameter:

none No entropy injection activated.

time RandomTime tactic only, with no analysis of the software under test.

program RandomInstruction tactic. The AddressRegions parameter is set to the entire address range of the program under test, which is statically linked. The Fraction parameter is set to 0.001 and Fallthrough is zero. ProcessThreshold is set to three, which makes Njord reset the breakpoints when `increment_n` has finished.[3]

function RandomInstruction tactic with same parameters as above but AddressRegions set to the instructions of `increment_n`.

regression RandomInstruction tactic with same parameters as above, but AddressRegions set to the two-instruction window in the load/increment/store sequence.

Statistics from simulated scenarios are shown in Figure 10. The column "E/R" shows the measured frequency of erroneous race closures per iteration.

The `none` strategy uses no entropy injection, and serves the purpose of verifying that races are no more likely to trigger in a simulated machine than in real machines.

The `time` strategy illustrates that unguided injection strategies are ineffective at provoking defects, even in a simple program, such as this.

The other strategies based on RandomInstruction are more useful strategies, and do manage to provoke errors with a reasonable probability. As expected, the probability of triggering a race increases as the breakpoint address range decreases.

The `program` strategy is a noise maker, and does not require the developer to guide the injection. It guesses that some statement anywhere in the program contains a race condition that can trigger if a delay is inserted. In spite of guessing breakpoints among more than 100,000 instructions, it manages to provoke a few races.

The `function` strategy illustrates the case where the programmer has a good hint on where to search, or where the developer has domain-specific knowledge about routines with high defect density.

[3] The pthreads implementation in SuSE Linux 7.3 creates an extra maintenance thread — hence three threads and not two.

injection	Delay	bkpts	S	code size	E / R	injections / R	cycles / injection	injection ratio
none	0	n/a	1	n/a	0	0	0	0
none	0	n/a	10	n/a	0	0	0	0
none	0	n/a	100	n/a	0	0	0	0
none	0	n/a	1K	n/a	0	0	0	0
time	10M	n/a	1	n/a	0	0.131	10M	0.5
time	10M	n/a	10	n/a	0	7.91	9972K	0.5
time	10M	n/a	100	n/a	0	8.07	10M	0.5
time	10M	n/a	1K	n/a	0	8.74	10M	0.5
time	100M	n/a	1	n/a	0	0.126	98M	0.5
time	100M	n/a	10	n/a	0	8.74	100M	0.5
time	100M	n/a	100	n/a	0	8.64	99M	0.5
time	100M	n/a	1K	n/a	0	9.12	99M	0.5
program	10M	105	1	421K	0.001	0.528	9236K	0.493
program	10M	105	10	421K	0.001	0.42	8855K	0.49
program	10M	105	100	421K	0	1.95	9328K	0.454
program	10M	105	1K	421K	0	2.21	9917K	0.213
program	100M	105	1	421K	0.001	0.449	89M	0.499
program	100M	105	10	421K	0.004	0.418	96M	0.499
program	100M	105	100	421K	0	1.96	94M	0.495
program	100M	105	1K	421K	0.001	2.05	94M	0.437
function	10M	1	1	296	0.02	0.756	10M	0.497
function	10M	1	10	296	0.025	0.728	9912K	0.496
function	10M	1	100	296	0.015	0.721	6715K	0.377
function	10M	1	1K	296	0	0.748	10M	0.104
function	100M	1	1	296	0.024	0.762	102M	0.5
function	100M	1	10	296	0.022	0.722	99M	0.5
function	100M	1	100	296	0.021	0.719	57M	0.483
function	100M	1	1K	296	0.021	0.735	65M	0.338
regression	10M	1	1	8	0.867	0.999	9625K	0.498
regression	10M	1	10	8	0.761	0.999	9769K	0.498
regression	10M	1	100	8	0.718	1	3727K	0.425
regression	10M	1	1K	8	0.04	1	9837K	0.152
regression	100M	1	1	8	0.917	0.999	103M	0.5
regression	100M	1	10	8	0.895	1	100M	0.5
regression	100M	1	100	8	0.896	0.999	19M	0.488
regression	100M	1	1K	8	0.863	1	25M	0.401

Figure 10: Race statistics, simulated machines with entropy injection. R = 1000 for all runs. Bkpts = number of breakpoints.

The `regression` strategy uses RandomInstruction tactic with same parameters as above, but AddressRegions set to the two-instruction window in the

load/increment/store sequence. It approximates the best possible strategy that one can hope to achieve in practice — a strategy that only injects delays in race windows. It is also an example of how a race condition regression test can be written. When a developer has discovered a race condition that involves one or more code statements, he can write a strategy that injects delays when those statements are executed. After verifying that the new strategy is capable of provoking the defect he found, he corrects the defect and adds the new test to the regression test suite. If another developer adds incorrect code that contends for the same resource without using proper synchronisation, it is now likely to be discovered during regression testing. The results above show that `regression` provokes races with a high probability, sufficient for regression testing, as long as the Delay value is high. Since the shared variable has a known address in this scenario, `regression` could also have been implemented by using read and write watchpoints instead of execution breakpoints. The implementation would have been similar, and the results identical. The use of execution breakpoints, however, is applicable to more scenarios.

If a developer has discovered a malign race condition, and an execution path that leads to it, he can also create more explicit regression tests by writing strategies that place breakpoints at statements that were executed before the race, probe application state, and provoke a specific interleaving pattern that triggers an erroneous race closure. This variant of regression testing is slightly more complex and will not be demonstrated here.

In the statistics, we can see that the frequency of races triggered goes down as the value of S increases, in particular for the Delay value of 10M. When the amount of work before the race window exceeds the Delay value, the injected delay can come to an end before the uncommon race closure occurs, even if a breakpoint was set at the right address. A high Delay parameter is probably preferable in most cases, but long delays injected outside race windows have a performance cost.

The average number of delays injected per iteration is shown in the "injections / R" column and the average number of cycles injected per injection in the column "cycles / injection." The average number of cycles / injection is usually smaller than the delay parameter value, since some delays are cut short when the number of threads drops below the value of ProcessThreshold. The last column shows the number of injected cycles divided by the total number of simulated cycles (counting both simulated instructions and cycles in disabled processors).

5.2.3 Performance

Simulation performance is somewhat important for entropy injection. Better performance allows for more test runs, which increases the probability of discovering a defect. Simics performs about two orders of magnitude slower than the host machine. It is also used for other purposes and is therefore flexible. There are many configuration parameters and control options selectable at runtime, and this flexibility comes at the cost of lower performance. There are simulators

or emulators that have more limited use cases and perform better, less than 10 times slower than the host, but we are not aware of any that are deterministic and supply the adequate programming interfaces to be useful for programmatic debugging and entropy injection. It is not known what performance a simulator that only supplied the necessary services would achieve.

Performance statistics for the simulated scenarios above, which were executed on the computer scaramanga (see Section 5.2.1) are shown in Figure 11. The column "simulated seconds" shows the number of seconds that elapsed in the simulated world. The column "simulated instructions" shows the number of instructions executed by the simulator, i.e. excluding cycles elapsed on disabled processors. The next column shows the number of seconds elapsed in the real world and the last column the quota between the previous two columns.

The performance of the `none` strategy represents the base Simics performance. Test setup incurs some overhead, and longer tests tend to achieve higher performance than short tests. The `program` strategy suffers from low performance, and all tests for `function` and `regression` takes approximately the same time to execute. This indicates that the cost of resetting the RandomTime tactic, i.e. setting and removing breakpoints limits performance. Nornir's virtual address breakpoint implementation is designed to avoid complexity and not for simulation performance. There are several simple ways to improve breakpoint performance, for example by adding a virtual memory translation cache. In this scenario, there is approximately a factor of five (for $S = 1000$) in potential performance gain before Simics's base performance is reached. We have decided, however, not to pursue optimisations that add complexity and state, since managing complexity and correctness are the major challenges in implementing an entropy injector.

The `time` strategy shows an interesting increase in number of instructions per second in comparison to simulations without entropy injection. This is probably caused by the Simics's idle loop detection; when one processor is stalled, the thread running on the other processor ends up waiting for its sibling to terminate, and the simulator detects that the processor is idle and skips ahead. This indicates that injecting a delay only has a small cost in simulation time; if a delay is injected in a thread executing the application's critical path, the other processors wait in the idle loop and are simulated at high speed. If a delay is injected in a thread that is not executing the critical path, the application execution time does not increase.

Although simulation performance is important, it is not crucial for the method's applicability. There will always be scenarios, for example systems with a few communicating low-end embedded processors, where simulation on high-end workstations is faster than real world execution. Likewise, there are scenarios at the other end of the system size scale, e.g. peer-to-peer applications with millions of nodes, that are unfeasible to simulate with existing simulation technology. The performance of an implementation determines how large systems it is applicable for. Moreover, it is only necessary to trigger a defect once in order to reproduce and debug the problem. During development of an application, the application test suite can be executed many more times than we

injection	Delay	S	simulated seconds	simulated instructions	host seconds	instructions / second
none	0	1	0.174	80M	1488	53K
none	0	10	0.334	134M	1495	89K
none	0	100	12.8	4316M	1518	2841K
none	0	1K	192	64G	1975	32M
time	10M	1	0.444	1315M	1521	864K
time	10M	10	464	78G	1980	39M
time	10M	100	478	81G	1979	41M
time	10M	1K	513	87G	2104	41M
time	100M	1	0.211	12G	1550	8007K
time	100M	10	5171	880G	6278	140M
time	100M	100	5049	857G	6179	138M
time	100M	1K	5311	903G	6485	139M
program	10M	1	29.4	5022M	96K	52K
program	10M	10	22.5	3871M	94K	41K
program	10M	100	119	21G	187K	116K
program	10M	1K	305	80G	200K	401K
program	100M	1	238	40G	98K	408K
program	100M	10	239	40G	92K	436K
program	100M	100	1111	188G	190K	992K
program	100M	1K	1311	248G	189K	1306K
function	10M	1	45.3	7660M	9979	767K
function	10M	10	43.2	7319M	9975	733K
function	10M	100	38.1	7997M	9750	820K
function	10M	1K	214	64G	10K	6055K
function	100M	1	465	78G	10K	7597K
function	100M	10	426	71G	10K	7039K
function	100M	100	254	44G	9933	4458K
function	100M	1K	421	93G	10K	8807K
regression	10M	1	57.4	9699M	10K	914K
regression	10M	10	58.3	9857M	10K	931K
regression	10M	100	26.1	5047M	10K	493K
regression	10M	1K	192	54G	10K	5043K
regression	100M	1	615	103G	10K	9503K
regression	100M	10	597	100G	10K	9172K
regression	100M	100	116	20G	10K	1946K
regression	100M	1K	192	38G	10K	3612K

Figure 11: Simulation performance.

have resources to do for the example in this article. Since test executions are independent, they can be run in parallel on farms of cheap commodity comput-

ers, and also take advantage of the predicted rapid increase in number of cores per processor over the coming years.

6 Related work

Detecting software concurrency defects has been a major challenge in computer science for a long time, but only marginal progress has been made. One of the thickest books on software testing by Binder [Bin99] spends one of its 1191 pages on the issue, concluding that race conditions and other concurrency errors are common problems, and suggests running multiple test processes concurrently and randomising process start intervals. This brief treatment illustrates the industrial state of the art practice.

There are various ad-hoc, application-specific methods involving stressful load, random thread suspensions, artificial resource consumption, etc. This is called *noise making* or *controlled execution*. A noise maker can either be an *irritator*, which steals resources by creating additional load on the system, or it can take control over one factor that is normally nondeterministic, either message arrivals or thread scheduling, and add noise to its behaviour. There are implementations of generic noise makers for message-passing systems [DKF93, DKF94] and for Java virtual machines [Sto02, EFG+03]. The noise added is either completely random, *white noise*, or guided by various heuristics [BAEFU06, Eyt05, SA06], for example shared variable access monitoring [BFU03, BAFE03] or coverage analysis [EFN+02]. Noise makers have similarities with entropy injectors, but since only one factor is controlled, full reproducibility is not achieved, and race conditions relating to other factors cannot be detected. Moreover, noise makers attempt to provoke defects with application-independent heuristics or state space search, whereas the primary purpose of entropy injection is to provide the ability to write guided regression test cases for race conditions.

There exists a few methods for detecting data races in shared memory programs. The most popular method is called dynamic race detection, and several research implementations [YRC05, AHB03, HR01, BHPV00] and some industrial strength implementations [Int, Val] exist. Dynamic race detectors monitor shared memory references and synchronisation operations made by the software under test. These are analysed according to the lock set algorithm [SBN+97], the vector clock algorithm [RB00], or a combination thereof. Pairs of concurrent load/store or store/store memory accesses that were not separated by synchronisation, and could have been executed in a different order are reported. Such pairs indicate nondeterministic program behaviour, and often indicate programming errors. Dynamic race detection does not require a comprehensive test suite, and is a cost-effective method for finding the defects that it is capable of detecting. Unlike entropy injection, however, it is only capable of detecting a limited class of race conditions in a particular homogeneous shared memory environment. Moreover, it only reports nondeterministic behaviour in possible execution paths that are close to the path that was executed. Dynamic race

detectors also tend to produce false positive reports, which must be manually inspected and suppressed by the user.

There are also static code analysis tools for detecting some concurrency errors [NAW06, PFH06, EA03, HJMS03, QW04], such as missing locks around shared variable accesses. Static analysis tools are even more limited in the types of errors they can detect than dynamic analysis tools and also produce false reports, but they are easy to use and integrate into the development process. Yu et al. and Havelund et al. have written good overviews of static and dynamic race detection tools [YRC05, HSU03].

Malign race conditions that appear during execution on real machines can be replayed and debugged with a runtime system that logs asynchronous events in sufficient detail to replay the full execution. This technique is called deterministic replay [LMC87], and can be performed at several abstraction levels, for example with hardware support [XBH03], virtual hardware [KDC05], specialised operating systems [TH00], process interception [GASS06], or in virtual machines [KSC00]. Deterministic replay requires intrusive support in the system under test, and it is usually only capable of replaying single nodes or homogeneous nodes in a distributed system, but exceptions exist [KSC00, GASS06]. Implementations for multiprocessor computers are rare [Dun06], and limited in functionality and performance, since logging cache-to-cache transfers is infeasible without hardware support. There are several good overviews on deterministic replay techniques [Sai05, DFD06, Hus02, CGC+03], and we will not mention all variations here.

Virtualisation programs, such as Xen [DFH+03], are similar in nature to simulators, but they are generally not isolated from nondeterministic input from the outside world, and can therefore not provide deterministic execution. Pervasive debugging, a method for deterministic debugging of distributed applications running in Xen domains [HHH04], can be regarded as a light-weight version of a subset of holistic debugging. Virtualisation achieves better performance than instruction-set simulation, but provides more limited observability and only supports distributed systems with homogeneous hardware.

Pervasive debugging and deterministic replay are less expressive debugging methods than holistic debugging. The implementations are typically focused on interactive debugging of a limited number of similar processes, and provide no support for automation. They do not include an abstraction translation stack, and either support observing a single abstraction layer, or in the case of the pervasive debugger, require modifications to the virtual machine of each process observed [Ho05]. These techniques are primarily useful for debugging problems that appear with high probability on real machines. Without entropy injection, they cannot increase the probability of triggering race conditions. A pervasive debugger could potentially be a platform for creating an entropy injector if it, in addition to standard interactive debuggers, supported programmable debugging services, which can be used by the injector.

Model checking is a method that can be used for verifying concurrency properties of distributed algorithms. Developers create state machine models of a system's processes, and specify system invariants. A model checker, for exam-

ple SPIN [Hol97], performs random system executions based on the models and reports violations of system invariants. If the state space is sufficiently small, a complete search can be made, or various formal methods can be applied on the model. The required extra work of creating models and ensuring that they match the implementation makes model checking cost-ineffective for most applications, but it is popular in academia and for some industrial scenarios, for example verification of communication protocols. The method, however, is restricted to detecting errors that are present in the state machine model, which is a simplified approximation of the real implementation, and it cannot detect lower level defects.

The use of simulation for quality assurance is an integrated part of the development process for almost all types of complex engineering products, with the exception of software products. Designs for cars, ships, buildings, bridges, and aeroplanes are always tested under simulated stressful conditions before construction. For these products, achieving the quality levels required today is either impossible or economically infeasible without simulation. In computer hardware engineering, testing with register transfer level (RTL) or gate level simulation is the main verification methodology. Concurrency errors are detected by injecting artificial timing variations in communication devices, without violating specifications. RTL simulation is very slow, between six and ten orders of magnitude slower than the host machine, and a high-end server has only been tested for a very short simulated time period before tape out. Yet, through careful crafting of test cases and timing variation injection, the quality level achieved in highly concurrent hardware systems is much higher than in concurrent software systems of comparable complexity.[4]

7 Discussion

Simulating a computer system under test and injecting random timing variations is a naïve, brute force approach to race condition detection, in contrast with traditional research approaches for concurrency error avoidance, which are based on various forms of intelligent program analysis or strict programming paradigms. Any program analysis, however, must assume properties of the analysed system, and therefore narrows the applicability to specific classes of systems. The author has striven to minimise the assumptions required for entropy injection, and entropy injection is designed to be applicable for all types of intermittent errors in all types of software systems. The example scenario above involves a threaded shared memory program running in a multiprocessor machine. This scenario was chosen because it is easy to create, set up in a simulation, and explain in an article. Entropy injection does not assume shared memory nor homogeneity, however.

The injection strategies demonstrated are deliberately kept simple in order to

[4]Unfortunately, the author has failed to find an appropriate literature reference for this claim. Innovation in practical hardware verification is driven by high-end hardware vendors, who do not publish their methods in academic press.

illustrate that even minimal dynamic program analysis is sufficient for provoking defects with high probability. More sophisticated program analysis or model checking can be added as injection strategies. It is likely that this would improve test efficiency, but possibly at the cost of generality. This is left for future work, along with many research questions: How well does entropy injection work for different types of real-world applications? How does one measure and reason about test coverage? Do generic entropy injectors work sufficiently well, or do we need application-specific injectors? Can we automate choices of injector parameters?

An entropy injector with an intelligent strategy may be able to unravel most malign race conditions in an application with a good test suite. Achieving high coverage in terms of race event interleavings, however, will require test engineering effort. Entropy injection may become a method that automatically finds some race conditions without developer guidance, but it should primarily be regarded as technology that enables application developers to explicitly control all factors affecting execution and thereby test for race conditions.

8 Conclusions

We present entropy injection, an incremental improvement over traditional test methods that enables application developers to test for arbitrary types of race conditions. An entropy injector is built on a simulation-based holistic debugger. The underlying simulator is deterministic and provides explicit control over factors that usually make test executions nondeterministic, such as process scheduling and time flow. Developers can use the holistic debugger to monitor application state and steer injection of timing variations to stress test sensitive program constructions, and to create regression tests for discovered concurrency errors. We have created a proof-of-concept entropy injector implementation and provide a simple demonstration of how it can be used for creating test cases for race conditions.

The author believes that entropy injection is a feasible method for addressing the difficult problem of detecting malign race conditions. It is practical, does not require new programming paradigms, and works with existing methods and processes for software development and quality assurance. It does require the development of a production quality holistic debugger and injector, and it requires test engineers to develop injection strategies. This seems like a reasonable price to pay for solving the increasingly pressing issue that no cost-effective method for detecting arbitrary race conditions exists.

References

[AHB03] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Journal on Software Testing, Verification and Reliability*, 13(4):220–227, 2003.

[Alb06a] Lars Albertsson. Holistic debugging. Technical report, Swedish Institute of Computer Science, August 2006.

[Alb06b] Lars Albertsson. Holistic debugging — enabling instruction set simulation for software quality assurance. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Monterey, California, September 2006.

[BAEFU06] Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. Producing scheduling that causes concurrent programs to fail. In *PADTAD '06: Proceeding of the 2006 workshop on Parallel and Distributed Systems: Testing and Debugging*, pages 37–40, 2006.

[BAFE03] Yosi Ben-Asher, Eitan Farchi, and Yaniv Eytani. Heuristics for finding concurrent bugs. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.

[BFU03] Marina Biberstein, Eitan Farchi, and Shmuel Ur. Choosing among alternative pasts. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.

[BHPV00] Guillaume Brat, Klaus Havelund, SeungJoon Park, and Willem Visser. Java pathfinder - a second generation of a java model checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.

[Bin99] Robert V. Binder. *Concurrency testing*, chapter 14.3.3, pages 744–745. Addison-Wesley, 1999.

[CGC+03] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. De Bosschere. A taxonomy of execution replay systems. In *In Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[DFD06] C. Dionne, M. Feeley, and J. Desbiens. A taxonomy of distributed debuggers based on execution replay. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques*, 2006.

[DFH+03] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[DKF93] Suresh K. Damodaran-Kamal and Joan M. Francioni. Nondeterminancy: testing and debugging in message passing parallel programs. In *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, pages 118–128, 1993.

[DKF94] Suresh K. Damodaran-Kamal and Joan M. Francioni. mdb: A semantic race detection tool for PVM. In *proceedings of the 1994 Scalable High Performance Computing Conference*, May 1994.

[Dun06] George W Dunlap. *Execution replay for intrusion analysis*. PhD thesis, University of Michigan, October 2006.

[EA03] Dawson R. Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

[EFG+03] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3–5):485–499, March/April 2003.

[EFN+02] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. *IBM systems journal*, 41(1), 2002.

[Ekm04] Magnus Ekman. *Strategies to Reduce Energy and Resources in Chip Multiprocessor Systems*. PhD thesis, Chalmers University of Technology, December 2004.

[Eyt05] Y. Eytani. Concurrent java test generation as a search problem. In *Fifth Workshop on Runtime Verification*, Edinburgh, UK, 2005.

[Gai85] Jason Gait. A debugger for concurrent programs. *Software Practice and Experience*, 15(6):539–554, June 1985.

[GASS06] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of USENIX*, pages 289–300, June 2006.

[GKO+00] Jeff Gibson, Robert Kunz, David Ofelt, Mark Horowitz, and John Hennessy. FLASH vs. (simulated) FLASH: Closing the simulation loop. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 49–58. ACM, November 2000.

[HHH04] Alex Ho, Steve Hand, and Tim Harris. PDB: Pervasive debugging with Xen. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (Grid 2004)*, November 2004.

[HJMS03] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer-Verlag, 2003.

[HM94] D. P. Helmbold and C. E. McDowell. A taxonomy of race detection algorithms. Technical Report UCSC-CRL-94-35, University of California at Santa Cruz, 1994.

[Ho05] Alex Ho. Personal communication, 2005.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[HR01] Klaus Havelund and Grigore Rosu. Monitoring Java programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[HSU03] Klaus Havelund, Scott D. Stoller, and Shmuel Ur. Benchmark and framework for encouraging research on multi-threaded testing tools. In *Proceedings of Parallel and Distributed Systems: Testing and Debugging*, Nice, France, April 2003.

[Hus02] Joel Huselius. Debugging Parallel Systems: A State of the Art Report. Technical Report 63, Mälardalen University, Department of Computer Science and Engineering, September 2002.

[Int] Intel Thread Checker.

[KDC05] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the USENIX 2005 Annual Technical Conference, General Track*, pages 1–15, 2005.

[KSC00] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed java applications. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pages 219–228, May 2000.

[LMC87] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The design and implementation of the 4.4 BSD operating system.* Addison-Wesley, 1996.

[MCE+00] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, February 2000.

[NAW06] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the Conference on Programming Language Design and Implementation.* ACM press, June 2006.

[NM92] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[PFH06] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM press, June 2006.

[QW04] Shaz Qadeer and Dinghao Wu. KISS: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–24, June 2004.

[RB00] Michiel Ronsse and Koenraad De Bosschere. Non-intrusive on-the-fly data race detection using execution replay. In *Automated and Algorithmic Debugging*, 2000.

[SA06] Koushik Sen and Gul Agha. jCUTE : Automated testing of multithreaded programs using race-detection and flipping. Technical Report UIUCDCS-R-2006-2676, University of Illinois at Urbana Champaign, 2006.

[Sai05] Yasushi Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, 2005.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[Sto02] Scott D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. Second Workshop on Runtime Verification (RV)*, Electronic Notes in Theoretical Computer Science, July 2002.

[TH00] Henrik Thane and Hans Hansson. Using deterministic replay for debugging of distributed real-time system. In *Proceedings of the 12th euromicro conference on real-time systems*, pages 256–272, Stockholm, June 2000.

[Val] Valgrind. http://valgrind.org/.

[XBH03] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the International Symposium on Computer Architecture*, June 2003.

[YRC05] Yuan Yu, Thomas Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.