Fast&Furious: A Tool for Detecting Covert Racing

Alberto Ros
Department of Computer Engineering
University of Murcia, Spain
aros@ditec.um.es

Stefanos Kaxiras
Department of Information Technology
Uppsala University, Sweden
stefanos.kaxiras@it.uu.se

ABSTRACT

Existing multi-threaded applications perform synchronization either in an explicit way, *e.g.*, making use of the functionality provided by synchronization libraries or in an implicit or "covert" way, *e.g.*, using shared variables. Unfortunately, the implicit synchronization constructs are prone to errors and difficult to detect.

This paper presents a tool that is able to detect implicit synchronization in multi-threaded applications. The detection is performed by ensuring that during the execution of an application under a memory model that provides sequential consistency for data-race-free applications (SC for DRF), every read returns the same value as if running under sequential consistency. If the previous condition is not fulfilled by the execution, the application has data races, which may be intended to perform implicit synchronization.

We analyze the applications in the Splash2 benchmark suite with the presented tool and we detect data races in 7 out of the 14 Splash2 applications. These data races perform implicit synchronization in all but one of the applications (6 out of 7). We analyze these implicit synchronization constructs and discuss their correctness.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Verification, Reliability, Languages

Keywords

Parallel applications, synchronization, data races, race detection

1. INTRODUCTION

PARMA-DITAM '15, January 19 - 21 2015, Amsterdam, Netherlands

http://dx.doi.org/10.1145/2701310.2701315

```
/* Initially X = Y = 0 */

X = 1;          |  Y = 1;
$r0 = Y;        |  $r1 = X;
```

Figure 1: Dekker's algorithm

Parallel programming is of increasing importance now that multicore architectures dominate the market. Unfortunately, parallel programming is difficult and prone to errors.

The memory consistency model, or simply memory model, defines what value a read should return in a multithreaded program. Arguably, the most intuitive memory model is sequential consistency (SC) [7]. A hardware provides SC if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and if the operations of each individual processor appear in this sequence in the order specified by its program. Despite being so intuitive, most processors do not adopt SC since its performance limitations can be dramatic. The reason is that it prevents both the software (compiler) and the hardware (processor) from reordering instructions.

Sequential consistency for data-race-free (SC for DRF) [3] is a memory model that provides sequential consistency for programs that do not have data races. Although not as intuitive as SC, it enables higher efficiency. In fact, Java and C++ have recently adopted a memory model that provides SC for DRF applications.

As defined by Sorin *et al.* [15], a data race occurs when two threads access the same memory location, at least one of the accesses is a write, and there is no intervening synchronization (even transitive) between the threads. This definition, therefore, assumes that the model distinguishes between synchronization and ordinary (non-synchronization or data) operations. In fact, languages like C++ include the declaration of `atomic` variables (or `volatile` in Java), to perform the synchronization among threads. If one considers all data races in a program as synchronization, the program would be data-race-free. For example, as mentioned by Adve and Boehm [2], to write Dekker's algorithm (Figure 1) correctly under the SC for DRF consistency model, one simply needs to identify the shared variables X and Y as synchronization variables.

However, applications do not always implement thread synchronization in an explicit or easy to detect way, *e.g.*, calling the lock, unlock, signal, broadcast, and wait functions provided by standard POSIX thread libraries or employing other synchronization libraries developed by programmers, as shown in Figure 2 (*explicit* synchronization

```
/* Initially X = 0 */

X = 1;                          |   WAIT(cond);
SIGNAL(cond);                   |   $r1 = X;
```

Figure 2: Explicit synchronization

```
/* Initially X = flag = 0 */

X = 1;                          |   while (flag == 0);
flag = 1;                       |   $r1 = X;
```

Figure 3: Implicit synchronization

from now on). Unfortunately, it is common to encounter programs that synchronize using *ad hoc* codes created by the programmer, as shown in Figure 3 (*implicit* or *covert* synchronization from now on). This kind of synchronization, which is commonly used for flexibility, performance reasons, or merely by non-expert programmers, is difficult to detect, difficult to debug, and prone to errors [19].

This paper presents Fast&Furious, a pin-tool [8] that checks that, an application presents the same behavior under the SC for DRF model as under the SC model during its execution. In essence, the tool checks that the value returned by every read in the application assuming SC for DRF is the same as the value returned assuming SC. In other words, the key characteristic of our tool is that is able to detect data races in the applications by emulating the SC for DRF model and expecting SC behavior, since data-race-free applications should provide such a behavior. (Section 2)

In applications where all synchronization is implemented in an explicit way (like in Figure 2), Fast&Furious will not find any data race during the execution of the application, since all reads will return an SC value. On the other hand, if the application has implicit synchronization or other data races, our tool can detect them if a read returns a non-SC value. In such a case, the tool shows the line of code that read the non-SC value and, optionally, the line of code that wrote the SC value. We categorize the applications that show SC violations as non-DRF. If the data race is intended to perform implicit synchronization, it should be either marked as synchronization (thus exposing it to the hardware) or re-implemented in a explicit way in order to be correct under the SC for DRF model.

We analyze all the applications from the Splash2 benchmark suite [18] with our pin-tool (Section 3). We consider as explicit synchronization every synchronization performed through the POSIX threads macros file provided along with the Splash2 benchmark suite. We found 88 data races in 7 out of the 14 Splash2 applications, 38 of them (in 6 of the Splash2 applications) being actually implicit synchronization. We analyzed the synchronization constructs and found that for some of them the behavior of the synchronization may not be the expected by the programmer.

Finally, we point out several research areas in which the proposed tool can be of great interest, such as methods that assume relaxed consistency models, accurate manycore simulation, debugging, and fence insertion in parallel applications (Section 4).

2. FAST&FURIOUS PIN-TOOL

This section describes the proposed tool, which is developed based on the PIN dynamic instrumentation tool [8]. Our tool checks that the value returned by every read performed by the application assuming a SC for DRF consistency model matches the value obtained assuming a SC model. To this end, the tool implements the behavior of both SC and SC for DRF consistency models.

The SC model is implemented assuming a unique shared memory structure that stores atomically each written value along with the written address. Every read will always observe the last written value, which is defined by the order in which the pin-tool interleaves the threads of the application during its execution.

The SC for DRF model allows reads and writes to be freely reordered, that is, all these reordering are allowed: R→R, R→W, W→R, and W→W. However, if these memory accesses are separated by explicit synchronization, reordering is not allowed. Instead of forbiding any reordering across synchronization points, our tool models acquire and release semantics for synchronization [5]. These semantics only order reads with respect to an acquire (ACQUIRE→R) and writes with respect to a release (W→RELEASE).

We instrumented the applications checked in this work to expose acquire and release semantics along with the explicit synchronization, which can be clearly identified, in the particular case of the Splash2 benchmarks, in the predefined POSIX thread macros. For example, synchronization in Figure 2, should have release semantics before the `SIGNAL` statement and acquire semantics after the `WAIT` statement. Otherwise, the read of the variable `X` could be performed before both `WAIT` and `X = 1`, leading to the unexpected result of zero.

In order to simulate the SC for DRF consistency model in our tool, we implement (unlimited) *private* caches per thread. Data blocks can be stored in the private caches without affecting other thread's caches (*e.g.*, causing downgrades or invalidations). Hence, reads and writes are performed locally to these caches, without propagation. This is equivalent to allowing any reordering of writes and reads (R→R, R→W, W→R, and W→W).

However, when a thread reaches a synchronization point with either acquire or release semantics, some reorderings are forbidden. Upon a synchronization point with acquire semantics all the contents in the private cache of that thread must be invalidated, and consequently cached data blocks, if accessed, must be re-fetched again. This is equivalent to ensuring ACQUIRE→R ordering. On the other hand, when the thread reaches a synchronization point with release semantics, writes performed in the cache must be performed by writing-back the value in the (unlimited) memory. This is equivalent to ensuring W→RELEASE ordering.

Our tool compares for every read in the application if the value obtained is the same under both consistency models. If it matches for all the reads, the application exhibits SC behavior under a DRF consistency model. Otherwise, the program contains implicit synchronization or other hidden data races. The tool then shows the line of code where the racy read happened and, optionally, at the expense of performance, the last line of code that modified the value. There is either a data race between those two lines of code, or an intervening implicit synchronization.

As an example, let's consider the racy code in Figure 3. Our pin-tool will detect two races: one for the variable `X` and one for the variable `flag`. Note that `X` is not actually a race under SC, because of the order enforced by `flag`. However,

a compiler may move `$r1 = X` before the while statement, since there is not a data dependence between them, thus converting it into a race. The problem of this example is that the synchronization requires release and acquire semantics. For example, acquire semantics would forbid hoisting the read of `X` before the read of `flag`.

In absence of any *implicit* synchronization, our tool allows any access reordering inbetween *explicit* synchronizations. We take advantage of this in the SC for DRF model by "hoisting" all reads of a thread above the writes of other threads. This is possible because, in this particular model, we implement infinite caches that are not affected by the outside, until an explicit synchronization is encountered.

Our ability to detect a race between a read and a write under these conditions depends on the three possible scenarios in SC: (i) the write always happens before the read (due to some implicit synchronization), (ii) the write always happens after the read (again due to some implicit synchronization), and (iii) the write may happen either before or after the read (there is not implicit synchronization to order them). In the first scenario, our tool will always detect the implicit synchronization. In the second scenario our approach does not detect a different value between the SC and the SC for DRF models, and therefore fails to identify the implicit synchronization. In the third scenario, our tool may or may not detect the race, depending on whether thread interleaving allows the write to be performed before the read in program order.

However, it is the first case (write before the read in SC) rather than the other two that is of interest (for detecting possible SC violations in the SC for DRF model). This is because in the third case, the SC model does not specify which value should the read return, and in the second case, in the SC for DRF model the read would never return the value of the write, which is the expected behavior: both the SC and the SC for DRF models result in the same read value for the variable.

In contrast to other tools such as Helgrind [17] or ThreadSanitizer [14], our tool does not show false positives in the sense that if it detects a data race, it is either a real race under SC or there is implicit (racy) synchronization between the write and the read. For example, in the false positive examples shown in ThreadSanitizer [14], our tool does not detect them because it considers acquire semantics for locks and release semantics for unlocks.

3. EXPERIMENTAL RESULTS

3.1 Methodology

We run our tool 100 times for each Splash2 application with the number of threads varying from 2 to 64. The input sizes are the recommended in Splash2 [18]. We can only detect races that can appear for the specific inputs. The applications are compiled with the -g flag in order to be able to identify the lines of code where races occur. The non-SC values reported by some executions under the SC for DRF model where analyzed to find the reason of the data race and are discussed in the following sections. It is useful to note that with a single run for each application and from 2 to 64 threads, our tool was able to detect 68 out of the total 88 data races that have been detected after the 100 runs per application at each thread count.

3.2 Detected implicit synchronization

Table 1 shows for each application, both the explicit and implicit synchronizations. For the explicit synchronizations we show the number of locks, barriers, and signal/wait found in the code (not the number of times that they were executed). If implicit synchronization (or data races) were not found by our tool, we consider the application as DRF. Otherwise, the data races that where found are classified as spin-loops, conditionals, or assignments.

The *Spin-loops* label implies that the racy read is the exit condition of a loop. This represents implicit synchronization in the code of the application, similar to a signal/wait or a barrier construct. The *conditionals* label reflects those races where the read is the condition of an `if` statement. For example, this can be the case of a thread that performs some work on a data structure only if it has been already created or updated by another thread, so they employ double-checked locking to reduce lock contention. Finally, *assignments*, are data races not involved directly in synchronization. They are either real data races, where the programmer may not care to get the last written value, or a write and a read at the opposite sides of implicit synchronization, as the variable `X` in Figure 3.

Our tool shows that half of the applications in the suite contain data races. In particular, 6 out of the 14 contain spin-loops or conditionals, in which the update of the condition variable is part of a data race, similar to the example shown in Figure 3, or double-checked locking.

Additionally, we have found subtle differences in the synchronization constructs with respect to ones described in Splash2 [18]. They do not match completely either with the explicit set of synchronization or with the explicit plus implicit set of synchronization reported in this work. For example, *Cholesky*, is supposed to have signal/wait synchronization, but in fact, it is implemented in an implicit way. However, other applications like *Barnes* or *FMM* have implicit synchronization that matches the signal/wait pattern, but such construct is not present in the description of the Splash2 applications.

Finally, neither our results nor the results obtained with the Helgrind tool [17], a data race detector part of the Valgrind binary instrumentation tool [11], match with the results presented in SyncFinder [19]. In particular, we did not find in the *FFT* application any implicit (*i.e.*, *ad hoc*) synchronization.

3.3 Volatile type and Synchronization in C/C++

We also found during our analysis that in most of the implicit synchronization constructs —in particular when using spin-loops— the programmer marks the condition variable as `volatile`. Before discussing this practice, we first describe the semantics of the `volatile` qualifier in C, as summarized by Regehr [12].

1. For every read from a volatile variable, the machine must load from the memory address corresponding to that variable.
2. For every write to a volatile variable, the machine must store to the corresponding address.
3. Accesses to volatile variables should not be reordered.

Although performing *ad hoc* synchronization with `volatile` variables is not a good practice [13], it is employed in spin-

Table 1: Synchronization in the Splash2 benchmarks

	Explicit				Implicit		
Benchmark	Locks	Barriers	Signal/Wait	Is DRF?	Spin-loops	Conditionals	Assignments
Barnes	6	6	0	No	5	5	23
Cholesky	7	4	0	No	1	1	0
FFT	1	7	0	Yes	0	0	0
FMM	28	13	0	No	9	3	19
LU-nc	1	5	0	Yes	0	0	0
LU	1	5	0	Yes	0	0	0
Ocean-nc	4	19	0	No	0	0	1
Ocean	4	20	0	Yes	0	0	0
Radiosity	37	5	0	No	1	7	6
Radix	1	7	3/2	Yes	0	0	0
Raytrace	12	1	0	No	0	2	0
Volrend	14	13	0	No	4	0	1
Water-Nsq	9	9	0	Yes	0	0	0
Water-Sp	10	9	0	Yes	0	0	0

loops found in Splash2 applications to enforce a load from memory on every check of the exit condition. Otherwise, the value of the condition can be kept in a register, which would not be updated with the value written by other threads, and consequently, the loop would spin infinitely. As an example, if in the code given in Figure 3, the `flag` variable is not qualified as `volatile`, the `while` loop may never exit.

But there is another issue with synchronizing by spinning on `volatile` variables, and it has to do with the reordering of accesses preformed either due to compiler optimization or run-time speculation. As discussed by Sutter [16], there is currently no convergence in the C/C++ compiler implementations about reordering ordinary reads and writes across `volatile` reads and writes. This is because the C++ Standard leads to different interpretations, and as a result, each compiler vendor has its own interpretation. For example, GCC, Intel CC, Sun CC, and Open64 compilers allow read and writes to move across `volatile` memory operations, even ordinary writes to volatile writes. On the other hand, other compilers such as LLVM and Microsoft C/C++ do not allow any reordering across accesses to `volatile` variables.

The consequence of these different interpretations is that, even assuming that the `flag` variable in Figure 3 is qualified as `volatile`, the code is not portable, since when compiled with, for example, GCC, the load of `X` can be moved before the `while` statement, thus being able to get the value of zero in `$r1`, which is not the intention of the programmer.

3.4 Details of races

This section analyzes the six applications where we found implicit synchronization, starting with the application with the least implicit synchronization, and ending with the one with most. We do not comment on assignment data races, since they are either: i) races where the programmer does not expect one thread to see the last value written by another thread, or ii) are a consequence of an implicit synchronization separating the read and the write (and therefore would disappear if the synchronization was made explicit).

3.4.1 Raytrace

Raytrace contains two races on conditionals in the file *workpool.c*. The synchronization skeleton of *Raytrace* is shown in Figure 4. It essentially extracts work items from a pool. This in an example of double-checked locking, which is not considered a good programming practice [9]. In fact, for this

```
GetJob(...) {
  LOCK(gm->wplock);
  w = gm->workpool[0];
  if (!w) {
    gm->wpstat[0] = 0;
    UNLOCK(gm->wplock);
    return 0;
  }
  gm->workpool[0] = w->next;
  UNLOCK(gm->wplock);
  ...
  return 1;
}
```

```
if (gm->wpstat[0] == 1)
  if (GetJob(...) == 1)
    ...
```

Figure 4: Implicit synchronization in Raytrace

```
LOCK(tasks[i].taskLock);
if (is_probe) {
  ...
  tasks[i].probeQ = t;
  ...
} else {
  ...
  tasks[i].taskQ = t;
  ...
}
UNLOCK(tasks[i].taskLock);
```

```
for (;;) {
  if (tasks[j].taskQ
      || tasks[j].probeQ) {
    LOCK(tasks[j].taskLock);
    if (tasks[j].probeQ) {
      ...
      tasks[j].probeQ = ...;
      ... }
    if (tasks[j].taskQ) {
      ...
      tasks[j].taskQ = ...;
      ... }
    UNLOCK(tasks[j].taskLock);
    ...
  } else
    while(!tasks[j].taskQ
        && !tasks[j].probeQ);
}
```

Figure 5: Implicit synchronization in Cholesky

synchronization to be correct, the conditions are re-checked inside a critical section of the thread extracting the work item from the pool, since otherwise the conditions can be simultaneously modified by another thread (the unprotected check is racing with a write).

3.4.2 Cholesky

The spin-loop and the conditional synchronization found in *Cholesky* corresponds to the same synchronizing construct in the file *fm.c* whose skeleton is shown in Figure 5.

A thread creates some tasks within a critical section, and the other thread checks their availability out of the critical section (both in the first `if` and in the `while` statements). If the `if` condition is true, the lock will be acquired and the

```
LOCK(Global->CountLock);
Global->Counter- -;
UNLOCK(Global->CountLock);
while (Global->Counter);
```

Figure 6: Implicit synchronization in Volrend

```
/* Initially X = 0 */

X = 1;                          | LOCK(Global->CountLock);
LOCK(Global->CountLock);        | Global->Counter- -;
Global->Counter- -;             | UNLOCK(Global->CountLock);
UNLOCK(Global->CountLock);      | while (Global->Counter);
while (Global->Counter);        | $r1 = X;
```

Figure 7: Undesired behavior for implicit barrier

task will be processed.

Again, this is an example of double-checked lock, but additionally, for the `while` loop not to spin forever, `taskQ` and `probeQ` are marked as `volatile`, which depending on the compiler used can prevent some optimizations, as described in Section 3.3.

This synchronization, although awkward, is correct. But in some systems it maybe be a long time until it sees the value of the spin-waiting variable, or even it could never see the current value, therefore, leading to a livelock.

3.4.3 Volrend

Volrend has four implicit synchronizations in the file *adaptive.c*. Two of these synchronizations correspond to barriers implemented with locks to decrease a counter inside a critical section and a `while` loop outside the critical section spinning on the counter until it reaches zero. Figure 6 shows the code.

Again, the `Counter` variable is declared as `volatile`. Although this prevents the `while` loop from spinning forever, even if another thread sets the value of `Counter` to zero, it could have a non desired behavior. Consider, for example, the following scenario. Figure 7 shows a possible execution, where `$r1 = X` is executed before the `while` loop (due to either compiler reordering or out-of-order execution). In this case, `$r1` can contain the value 0 at the end of the execution.

The other two implicit synchronizations found in *Volrend* correspond to the same synchronization construct which, in essence, acts as a barrier similar to the one discussed previously. The same issues, found in the previous barrier, apply.

3.4.4 Radiosity

Radiosity implements a barrier by increasing a counter and waiting until all threads arrive to the barrier. However, in the mean time, it checks for some work to do. If a thread finds some work to do, it decrements the barrier counter and processes the work. When it finishes, the counter is increased and the thread waits again for the other threads to arrive. The counter is not qualified as `volatile` when loading it to check the exit condition in the while statement, but is temporally qualified as `volatile` inside the while. In order to prevent dangerous reordering of accesses, there is an explicit barrier at the end of this implicit barrier. *Radiosity* contains also some racy conditionals that are, again, examples of double-checked locking.

3.4.5 Barnes

Barnes concentrates most of its races in the file *load.c*, although it also has one race in the exit condition of a `for` loop

```
while(!(r->done)) {
  /* wait */
}
```

Figure 8: Implicit synchronization in Barnes

```
while (b->interaction_synch != b->num_children) {
  /* wait */
}
```

Figure 9: Implicit synchronization in FMM

and two assignment races in *code.c* and *grav.c* respectively. In `load.c`, *Barnes* has three spin-loops and four conditionals that race. In two of the loops, the racy load is in the exit condition of a `for` loop. The racy variables in the `for` loops are not qualified as `volatile`.

The other loop of *load.c* corresponds to a busy waiting construct, as shown in Figure 8. In this case, `done` is qualified as *volatile*, but again, as in Figure 7 a subsequent load can be executed before the while statement, thus possibly returning an undesirable value.

3.4.6 FMM

FMM has multiple implicit spin-loops in *construct_grid.c*, *cost_zones.c*, and *interactions.c*, similar to the one in Figure 9. Again, even with the condition variables qualified as `volatile`, this code can produce undesired behaviors due to reordering of memory accesses. It also has double-checked locking constructs in the files *construct_grid.c*, *cost_zones.c*, and *partition_grid.c*.

4. APPLICABILITY

The Fast&Furious tool presented in this work has applicability in several research areas. This section describes its applicability in the most relevant areas.

4.1 Checking applications under weak memory models

Some memory models such as Release Consistency (RC) [5] or SC for DRF [3] rely on exposing synchronization to the hardware. For applications with implicit synchronization (or data races) these models do not provide SC. This tool identifies implicit synchronization by detecting if a read returned a non-SC value, thus probably not following the programmer's intention and being able to cause even livelocks (*e.g.*, spinning forever in the `while` statement in Figure 3). It is also important for this models to propagate the racy accesses in a fast way. For example, the write that releases a lock should be seen by the acquire as fast a possible to reduce the acquire waiting time. Our tool identifies such races which could be marked for efficiency as fast-propagating.

4.2 Large-scale and accurate simulation

New methodologies for simulating manycore architectures suggest the usage of binary instrumentation tools [10]. When the consistency model simulated is weaker than the one provided by the host system, it is hard to ensure the correct behavior of the application. By attaching our tool to this simulation methodology the correctness of the executions evaluated in the simulation can be ensured. Additionally, having explicit synchronization is desirable for accurate trace simulation, as discussed by Goldschmidt and Hennessy [6].

4.3 Debugging and fence insertion

Our tool is able to isolate data races, where most of the concurrent bugs lie. Particularly, in this work we already inspect the isolated synchronization constructs and found non-desirable behaviors. Additionally, the synchronization skeletons found can be used as input in tools that provide optimal fence insertion for relaxed memory models, such as Memorax [1] which can only scale up to a small number of lines of code. Once all required fences are found, the application code can be fenced.

5. CONCLUSIONS

We developed a pin-tool that checks if the applications provide SC under the SC for DRF model. If SC is not provided, the program contains data races. We analyze the data races, and classify them. We discuss races that are used for implicit synchronization of threads. We found that almost half of the Splash2 applications contain data races.

This work leads to several interesting future directions, apart from the cases already discussed. For example, our tool does not ensure that an application is DRF under every possible interleaving. A combination with tools like Concurrit [4] would allow us to force other thread interleavings and discover all possible races during the execution, at least for a given input and a given thread count.

6. ACKNOLEDGMENTS

This work was supported by the "Fundación Seneca-Agencia de Ciencia y Tecnología de la Región de Murcia" under grant "Jóvenes Líderes en Investigación" 18956/JLI/13, and by by the Spanish MINECO, as well as European Commission FEDER funds, under grant TIN2012-38341-C04-03.

7. REFERENCES

[1] P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine. Memorax, a precise and sound tool for automatic fence insertion under tso. In *19th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 530–536, Mar. 2013.

[2] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, Aug. 2010.

[3] S. V. Adve and M. D. Hill. Weak ordering – a new definition. In *17th Int'l Symp. on Computer Architecture (ISCA)*, pages 2–14, June 1990.

[4] T. Elmas, J. Burnim, G. Necula, and K. Sen. Concurrit: A domain specific language for reproducing concurrency bugs. In *34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 153–164, June 2013.

[5] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th Int'l Symp. on Computer Architecture (ISCA)*, pages 15–26, June 1990.

[6] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *1993 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 146–157, May 1993.

[7] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers (TC)*, 28(9):690–691, Sept. 1979.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 190–200, June 2005.

[9] S. Meyers and A. Alexandrescu. C++ and the perils of double-checked locking: Part i. Dr. Dobb's Journal, `http://www.drdobbs.com/cpp/c-and-the-perils-of-double-checked-locki/184405726`, July 2004.

[10] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi. How to simulate 1000 cores. *Computer Architecture News*, 37(2):10–19, July 2009.

[11] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 89–100, June 2007.

[12] J. Regehr. Nine ways to break your systems code using volatile. `http://blog.regehr.org/archives/28`, Feb. 2010.

[13] A. D. Robison. Volatile: Almost useless for multi-threaded programming. Intel developer zone, `https://software.intel.com/en-us/blogs/2007/11/30/volatile-almost-useless-for-multi-threaded-programming/`, Nov. 2007.

[14] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Workshop on Binary Instrumentation and Applications (WBIA)*, pages 62–71, Dec. 2009.

[15] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[16] H. Sutter. Volatile vs. volatile. A tale of two similar but different tools. Dr. Dobb's Journal, `http://www.drdobbs.com/parallel/volatile-vs-volatile/212701484`, Jan. 2009.

[17] Valgrind-project. Helgrind: A data-race detector. `http://valgrind.org/docs/manual/hg-manual.html`, 2007.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[19] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *9th USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, pages 1–8, Oct. 2010.