Splash-3: A Properly Synchronized Benchmark Suite for Contemporary Research

Christos Sakalis
chrissakalis@gmail.com

Carl Leonardsson
Uppsala University
carl.leonardsson@it.uu.se

Stefanos Kaxiras
Uppsala University
stefanos.kaxiras@it.uu.se

Alberto Ros
Universidad de Murcia
aros@ditec.um.es

Abstract—**Benchmarks are indispensable in evaluating the performance implications of new research ideas. However, their usefulness is compromised if they do not work correctly on a system under evaluation or, in general, if they cannot be used consistently to compare different systems.**

A well-known benchmark suite of parallel applications is the Splash-2 suite. Since its creation in the context of the DASH project, Splash-2 benchmarks have been widely used in research. However, Splash-2 was released over two decades ago and does not adhere to the recent C memory consistency model. This leads to unexpected and often incorrect behavior when some Splash-2 benchmarks are used in conjunction with contemporary compilers and hardware (simulated or real). Most importantly, we discovered critical performance bugs that may question some of the reported benchmark results.

In this work, we analyze the Splash-2 benchmarks and expose data races and related performance bugs. We rectify the problematic benchmarks and evaluate the resulting performance. Our work contributes to the community a new sanitized version of the Splash-2 benchmarks, called the Splash-3 benchmark suite. [1]

I. Introduction

Among benchmark suites for the evaluation of parallel architectures, Splash-2 [1] stands out as the first with a wide variety of complex parallel applications (not simply numerical kernels) and continuous use for two decades. Newer benchmark suites, such as PARSEC [2], are gaining in visibility, albeit without replacing Splash-2 but rather complementing it, as attested by the many studies that use both newer suites *and* Splash-2.

Splash-2 was created in the context of the Stanford DASH project [3] as a means to standardize benchmarking in the then emerging large-scale cc-NUMA architectures. Emphasis in Splash-2 optimizations was given towards enhancing scalability. Among the many optimizations found in the Splash-2 applications, we focus on the ones that concern synchronization.[2] Splash-2 applications are written in C (some ported to C from FORTRAN) and synchronization in C programs, by virtue of C's low-level nature, closely interacts with memory model semantics.

[1]**The Splash-3 source code is available at** http://splash3.argodsm.com

[2]Other work [4] contributed fixes and enhancements to the original Splash-2 but without going into the synchronization structure of the applications.

At the time the Splash-2 was created, the C language standard did not specify a memory model. Lacking a clear C-language memory model, Splash-2 were written with stricter memory models in mind, such as the Sequential Consistency (SC) [5] or Total Store Order (TSO) [6] memory models. However, the latest C standards [7] define a memory model that guarantees sequential consistency for *data-race-free* (*SC for DRF* [8]) applications. Other high-level language standards (e.g., C++11 [9], [10], Java [11]) have also adopted a memory model that provides SC for DRF applications. In this model, applications containing data races may exhibit unexpected behavior.

As recently shown, some of the Splash-2 applications do contain data races [12]. These races exist in Splash-2 not because they are outright bugs but as *synchronization optimizations*, presumably in an effort to enhance scalability for a generation of machines where synchronization (e.g., locks) was deemed particularly expensive. Data races can lead to unexpected and often incorrect behavior when the applications containing them are used in conjunction with contemporary C compiler (that supports SC for DRF) or hardware (either simulated or real) that supports a more relaxed memory model than TSO, as for example Release Consistency (RC) [13] or a Weak memory model (e.g., ARM [14] or Power [15] architectures).

Unexpected behaviors translate to non-deterministic or incorrect output, or even to performance bugs. Performance bugs, in particular, can jeopardize trust on previous evaluations of the troublesome applications. For example, our work shows that two Splash-2 applications, precisely because of the interaction of the data races in them with contemporary C compilers or with weakly-ordered hardware, can perform erratically. The first example is *Radiosity* where, because of a race condition, it is possible that only one thread ends up with all the work (while there are many other idling threads), artificially increasing load imbalance (thus increasing run-time and ruining scalability) in arbitrary configurations. The second example is *Volrend* where, because of a data race, the same task can be redundantly executed by more than one thread, again artificially increasing run-time in arbitrary configurations.

To analyze Splash-2 we developed our own tool to check for SC for DRF compliance [16]. The tool detects data races at run-time and although it cannot be used to prove DRF correctness of programs (for this, other—perhaps more formal—approaches are needed), it has proven invaluable in pointing us to the troublesome spots in the source code for the applications that did exhibit data races. We found seven applications, out of 14 in the Splash-2 benchmark suite, with data races (Section III).

We then proceeded to address the trouble spots. We followed the least invasive strategy we could to remove the data races, aiming to deviate as little as possible from the functionality of the original code. Where necessary we include additional synchronization either with locks (lock-/unlock) or with conditional variables (signal/broadcast/wait). In the process, we discovered critical performance bugs in two applications that are perhaps the result of the initial effort to optimize synchronization in the original application (Section IV). The result of this process is a new version of the applications that is compliant with the contemporary C-standard memory model.

Finally, we characterize the new version with respect to performance, scalability, and network traffic, using detailed simulation of multiprocessor systems (Section V). Our modifications affect the execution time of some of the applications, but without affecting the average. We also report on the additional synchronization introduced to remove data races. The end result is a more robust benchmark suite that we release as the Splash-3 benchmark suite.

To summarize, our **contributions** are:

- We systematically check Splash-2 with respect to its synchronization optimizations. We uncover data races using thorough code inspection and employing tools we developed for this purpose (akin to the Fast&Furious tool [12]).
- We expose unknown performance bugs in Splash-2 applications and show how these bugs could affect previous simulations using these applications.
- We remove data races in the applications, adding synchronization where necessary, and make them compliant with the current C standard memory model. This makes the applications data-race-free, by *library synchronization primitives* only. Where necessary we change the code to fix performance or correctness bugs.
- We quantify the performance, traffic, and synchronization of the new applications using a simulation methodology that models in detail the memory hierarchy of multiprocessors.
- We offer to the community a more robust suite of applications, the Splash-3 benchmark suite.

II. Data Races

As defined by Adve and Hill [8], a data race occurs when two threads access the same memory location, at least one of the accesses is a store, and there is no intervening synchronization (even transitive) between the threads (see Sorin *et al.* for an authoritative tutorial [17]). What constitutes synchronization depends on the underlying memory model. 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` (or `atomic` in C++, `volatile` in Java) variables, to perform synchronization loads and stores among threads, in conjunction with other synchronization mechanisms such as locks and memory fences. If one considers all data races in a program as synchronization, the program would be data-race-free.

Why are, however, data races such a problem, especially for the Splash-2 suite? If we take, as an example, the popular x86 architecture, which specifies a model that is stricter than release consistency (TSO), we will find that there is a lot of software written for it that contains data races but has been working correctly for many years. However:

- While software written with TSO in mind has indeed been working for many years now, it is impossible to use them on any other platform that requires some more relaxed memory model. In the case of Splash-2, this limits researchers in using the benchmark suite for evaluating their modern designs.
- One of the reasons why software has been working so far, and particularly C/C++ software, is that compiler writers have been taking into consideration the stricter memory model of x86 (and presumably other platforms as well) and have been avoiding optimizations that might break their users' code. However, with the introduction of the C/C++ memory models in their respective standards, as well as their respective thread support libraries, and by declaring data races as undefined behavior, the compiler writers are now free to apply any optimizations they see fit. This means that programs that do not follow the memory model will start (and already have, as we will see in Section IV) exhibiting bugs and other unwanted behavior.
- An extension to the previous point, undefined behavior in an application leads to both unpredictable and inconsistent run-time performance characteristics.

III. Detecting the Data Races

In order to detect data races in the Splash-2 applications, and in conjunction with manually analyzing the code, we implemented a tool using Intel's PIN [18], a dynamic binary instrumentation framework. Our tool is an extension to the Fast&Furious tool [12]. In particular, the tool implements a memory model which is the same as the one defined in the C standard. As we mentioned, this model provides sequential consistency for data-race-free applications. The tool checks the values obtained at every load operation and compares them to the values that should have been obtained with an SC model. If the values coincide, the tool concludes that no data races where found during the execution of the application.

Otherwise, if inconsistencies in the value of the loads are found, the tool alerts about the data race, since the application should have an SC behavior *in absence of data races*. The tool therefore prints the line of code of both the inconsistent load and the last store to that address.

Although this run-time tool cannot verify DRF application correctness, it can at least ensure SC-execution of the applications. Hence, we can guarantee that during the simulations performed in this work no data races occurred, or if they did occur they did not affect SC semantics. In addition to that, we perform a large number of runs in an effort to find some of the more rare races. With the detailed information from the tool, we analyzed the races in the code and fixed them following the least invasive strategy we could in order to respect the intended behavior of the application.

Our goal for this release of Splash-3 is to have a benchmark suite that can be used for simulation, with the widely used input parameters that the original Splash-2 paper [1] describes. As such, we only checked and corrected the applications for those parameters and not for every possible input. The latter would be impossible to check dynamically at run-time, since it is an NP-complete problem.

Out of the 14 applications in the Splash-2 benchmark suite, seven were found to contain data races. These applications are *Barnes*, *Cholesky*, *FMM*, *Radiosity*, *Raytrace*, *Ocean-nc*, and *Volrend*. The data races can be classified in the following (*not* mutually exclusive) categories:

Double-checked locks (DCL), (a.k.a. conditional synchronization) which happen when a conditional statement (e.g., an `if` statement) that needs to be checked within a critical section is first checked outside the critical section. This is done in order to avoid acquiring the lock for the critical section if the check fails. This is generally not considered a good programming practice [19].

Spin-loops, almost always coupled with **`volatile`** variables, are meant as a lightweight implementation of signal/wait. Essentially, one or more threads will spin in a `while` loop waiting for some shared variable (usually qualified as `volatile` — it can either be a flag or it might contain actual data) to get some specific value, which is written by another thread. This is not correct, since the only ordering `volatile` guarantees is that different `volatile` accesses will not be reordered *in the same thread* [20]. In our modifications, we removed *all* volatile qualifiers from the applications and we replaced them with proper synchronization mechanisms. Namely, the most common way of implementing signal/wait is by using conditional variables [21] (also known as "monitors").

Fuzzy barriers, usually coupled with job stealing, is a synchronization mechanism where one or more threads reaches a barrier, but instead of just waiting there, they also try to find and perform some useful work. The problem with these barriers is the underlying synchronization which is often implemented in a racy manner.

Conditional statements, which are often used to implement the three constructs named above, are data races where the value obtained from a racy load is used to decide the result of a conditional statement, such as an `if` or a `while` statement.

"Benign" data races are these data races that are seemingly safe to perform, even under a relaxed memory model. They are usually (but not always) racy loads and stores to shared data (a.k.a. **racy assignments**) that is not important to keep fully coherent. Practically, all of the races mentioned above can be characterized as "benign". The name "benign" comes from the fact that they are seemingly safe, however, in reality it is possible that they might cause erroneous behavior [22], [23], [24], [25].

IV. The Data Races: Implications and Solutions

In this section we discuss the data races and bugs that we found in the Splash-2 benchmark suite, as well as their implications for the execution of the applications, and how we fixed them. We present only the most interesting code samples; the complete set of modifications can be found by comparing the source code of Splash-3 with Splash-2.

Generally, we try to keep our modifications as simple as possible, for two reasons: First, introducing complex synchronization mechanisms would hinder other researchers from using the applications, since they might not be available on every system. Second, we did not want to introduce unnecessary changes to the behavior of the applications. After all, one of the reasons why Splash-2 is so widely used is simply because its behavior is well know and has been extensively documented.

For these reasons, apart from requiring additional locks, the only new synchronization mechanisms that we have introduced are conditional variables (signal/broadcast/wait) [21] and also one release memory fence in *Barnes*.

A. Barnes

Barnes contains a double-checked lock in the `loadtree` function in the `load.C` file, which we fixed by removing the unprotected conditional statement. Also, it contains some unsynchronized spin-loops in the `hackcofm` function (same file), which we replaced with conditional variables. The conditional variable will unlock the lock that the thread is holding and then it will put the threads to sleep. When the condition is met, another thread will signal the variable, which will awake the thread and relock the lock. The interesting part in *Barnes* is that both the loads and stores to the variable of the spin-loop were done without synchronization, probably because it is a simple `volatile` flag variable. However, this means that the compiler is free to reorder both the loads and the stores in the code as it sees fit, which will cause incorrect behavior.

Except for these races, *Barnes* also contains a couple of other "benign" data races, the first one regarding the positions of the particles, and the second one regarding the parent of a node. Neither of these are particularly interesting, in the sense that *for now* the compiler seems to do the correct thing. However, since in the C standard races are considered bugs, we fixed these races by protecting the accesses with the appropriate locks.

Finally, we found a data race that can lead to a race condition and incorrect execution, in the `loadtree` function mentioned earlier. The skeleton of the racy code is shown in Figure 1. Consider two threads **A** and **B**, as well as a tree with just the root **R** and a leaf **L**, containing the maximum number of bodies allowed in a leaf. Thread **A** enters the `loadtree` function with `mynode = &R` and `qptr = &R.child[0] (=L)` (Lines 1, 2), acquires the lock for **R**, enters the leaf case (Line 10), and calls `SubdivideLeaf` (Line 12). In `SubdivideLeaf` **A** allocates a new empty node **N** and it sets (i) the first child pointer of **N** (`N.child[0]`) to point to **L**. Then, after returning from `SubdivideLeaf`, **A** sets (ii) `*qptr = &N` (same line). Now assume that (ii) happens before (i). Of course no x86 CPU will do that, but a standard compliant compiler might, as it is allowed under the current C memory model. Now let us assume that before (i) happens, thread **B** enters `loadtree`, with `mynode = &R` and `qptr = &R.child[0] (=N)` (Lines 1, 2). **N** is neither a leaf nor `NULL`, so **B** proceeds to `mynode = &N` and `qptr = &N.child[0] (=L)` (Lines 17, 18). Now **B** acquires the lock for **N**, enters the `NULL` case (Line 7), allocates a new leaf node **L'**, adds the new particle to **L'** and assigns `*qptr = &L'`. Now (i) takes place, overwriting the value of the first child pointer of **N** from **L'** to **L**, thus making **L'** and its particles disappear. In order to fix this issue, we inserted a release fence between the call to `SubdivideLeaf` and the assignment to `*qptr` in order to prevent the reordering of (i) and (ii). Usually the locks protecting the access to the variables will also protect from such reorderings, but in this case the fine grained locking prevented that. We avoided adding additional locks to prevent deadlocks caused by the overlapping locking pattern. This is an excellent example on how subtle and hard to find such bugs in the code can be.

```
1  nodeptr mynode = root;
2  nodeptr *qptr = &mynode->child[...];
3  ...
4  while(...) {
5    ...
6    LOCK(mynode->lock);
7    if (*qptr == NULL) {
8      le = InitLeaf(...); le->body[0] = ... ;
9      *qptr = le;
10   } else if (Type(*qptr) == LEAF) {
11     if (*qptr->num_bodies == MAX_BODIES_PER_LEAF) {
12       *qptr = SubdivideLeaf(...)
13     } else { ... }
14   }
15   UNLOCK(mynode->lock);
16   ...
17   mynode = *qptr;
18   qptr = &mynode->child[...];
19 }
```

Fig. 1: The code containing the data race bug in Barnes

```
1  LOCK(tasks[i].taskLock);
2  if (is_probe) {
3    ...
4    tasks[i].probeQ = t;
5    ...
6  } else {
7    ...
8    tasks[i].taskQ = t;
9    ...
10 }
11 UNLOCK(tasks[i].taskLock);
```

(a) Enqueue

```
1  for (;;) {
2    if (tasks[j].taskQ || tasks[j].probeQ) {
3      LOCK(tasks[j].taskLock);
4      if (tasks[j].probeQ) {
5        ...
6        tasks[j].probeQ = ...;
7        ... }
8      if (tasks[j].taskQ) {
9        ...
10       tasks[j].taskQ = ...;
11       ... }
12     UNLOCK(tasks[j].taskLock);
13     ...
14   } else
15     while(!tasks[j].taskQ && !tasks[j].probeQ)
16       ;
17 }
```

(b) Dequeue

Fig. 2: Implicit synchronization in Cholesky

B. Cholesky

In *Cholesky* the data races appear in the implementation of a task queue (Fig. 2). In particular, *Cholesky* utilizes conditional DCL synchronization (Line 2 in Fig. 2b) and a spin-loop (Line 15 in Fig. 2b), both in the task dequeue operation. First, the availability of a new task is checked without acquiring any locks, in an effort to avoid unnecessary locking if no tasks are available. Then, if no new tasks are available, the thread will spin on the head pointer of the queue until a new task is enqueued by another thread.

As both races are related, we fixed both of them at the same time. We removed the first conditional statement and we also moved the spin-loop inside the critical section, replacing it with a conditional variable (wait). The conditional variable releases the lock and suspends the spin-loop while waiting for another thread to signal that a new task has been enqueued. The enqueue function was modified accordingly, in order to signal the conditional variables waiting for the new tasks. The signal is placed inside the critical section in Figure 2a.

C. FMM

FMM has multiple spin-loops in the `cost_zones.C`, `construct_grid.C`, and `interactions.C` files. All of them are used in place of conditional variables, and since they are unsynchronized they cause a large number of other data races as well. Much like all the other places where we encountered similar patterns, we introduced conditional

```
1  void process_tasks(...) {
2     ...
3     t = DEQUEUE_TASK(...);
4  retry_entry:
5     while(t) {
6       switch(t->task_type) {
7         ...
8       }
9       t = DEQUEUE_TASK(...);
10    }
11    ...
12    while(global->pbar_count < n_processors) {
13      if(_process_task_wait_loop())
14        break ;
15      t = DEQUEUE_TASK(...)
16      if(t) {
17        LOCK(global->pbar_lock);
18        global->pbar_count-- ;
19        UNLOCK(global->pbar_lock);
20        goto retry_entry ;
21      }
22    }
23    BARRIER(global->barrier, n_processors);
24 }
```

(a) Barrier loop

```
1  long _process_task_wait_loop() {
2     finished = 0 ;
3     for(i = 0; i < 1000 && !finished ; i++) {
4       if((i & 0xff) == 0)
5         if((volatile long)global->pbar_count
6           >= n_processors)
7             finished = 1 ;
8     }
9     return(finished);
10 }
```

(b) Backoff function

Fig. 3: The fuzzy barrier implementation in Radiosity

variables for both the load and the store operations, thus fixing all the races.

In addition to these races, *FMM* also contains a number of "benign" racy assignments when accessing the `type`, `children`, and `num_particles` variables of the boxes, in the `construct_grid.C` and `partition_grid.C` files. While they are not quite used as implicit synchronization mechanisms[3], reordering how these variables are stored or loaded would cause the program to misbehave. For example, it is possible to read the wrong number of children before the child pointers have been updated, since the accesses are done completely unprotected. Surrounding the offending code with locks prevents any such reorderings.

Finally, there are a number of racy assignments when accessing the expansion terms, in the `interactions.C` file. Presumably those races exist because there is no need for complete correctness when reading the expansion values, and the programmer(s) wanted to avoid unnecessary lock contention. As always, we fixed them by introducing the necessary locking.

D. Radiosity

In *Radiosity* we found three different cases of data races. The first case, and the most interesting one, is the "fuzzy" barrier implementation.

Radiosity implements a barrier by increasing a counter and waiting until all threads arrive to the barrier. However, in the mean time, it checks if there is any work to do. This is implemented by polling both the barrier counter and the task queue inside a while loop. A simple backoff function (from the queue) is also used (called in Line 13 in Fig. 3a and defined in Fig. 3b). 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 once again waits for the other threads to arrive. The counter is not declared as `volatile`, but is temporarily qualified as `volatile` inside the backoff function (Line 5 in Fig. 3b). After the fuzzy barrier, there is also an explicit normal barrier, presumably to prevent either instruction reordering or threads exiting the fuzzy barrier prematurely. The data races in this case happen when the barrier counter is checked in two places without acquiring any locks, while other threads might be updating it.

This case is particularly interesting because here we can find an example of the **compiler doing something unwanted**. Specifically, when compiling with `gcc-5.2.0` or `clang-3.6.1` the compiler will **optimize away the backoff function** completely. The compiler sees only a loop checking a variable *without any synchronization* and assumes that the variable is not modified from any other threads. At the same time, the cast to `volatile` is discarded by the compiler, since according to the C standard [7] (6.5.4 "Cast Operators" § 5) a cast does not produce an lvalue and thus casting to a qualified type is the same as casting to the unqualified version of the type. This means that all the compiler sees is a loop (which is guaranteed to finish after a set number of iterations) that reads the same variable multiple times, without the variable being modified anywhere. Thus, for the compiler, it makes perfect sense to remove the loop altogether. As we explain in the introduction, before the new C standard and accompanying memory model, compiler writers used to be hesitant to perform such optimizations, but nowadays they are becoming more and more prevalent. For example, this optimization will not happen with the older `gcc-5.1.0`. In order to fix these issues, we protected all the accesses to the barrier counter variable with the preexisting appropriate lock.

The second case of data races is once more that of double-checked locking in the task queue. When trying to dequeue a task, the number of tasks in the queue is checked outside the critical section, in an attempt to avoid unnecessary locking if the task queue is empty. We know how DCL can be harmful, so we removed the conditional checks outside the critical section. In a similar manner, the length of the queue is also checked for load balancing reasons, again without any synchronization. We rectified this using the preexisting appropriate locks.

Finally, the third case is a number of unsynchronized

[3]For one, eliminating those races does not eliminate any other races that we found.

accesses to the radiosity values of the elements. Presumably, due to the iterative nature of the algorithm, reading an inaccurate radiosity value is not an issue, and the programmer(s) wanted to avoid unnecessary synchronization. While these accesses are not used for synchronization, and thus the possibility of them introducing bugs into the application is somewhat lower, we know that *every* data race is considered a bug in the C standard, so we fixed them by protecting them with the appropriate locks.

Except for the data races, we have also identified a race condition, in the same fuzzy barrier, which can lead to bad performance and scalability. Assume two threads (two for simplicity; this is worse with more threads) **A** and **B**. **A** finishes its work and reaches the barrier, incrementing the counter and spinning while waiting for **B** to finish as well (Lines 12 and so forth). Then, while **B** is processing its current task, it creates a new task and enqueues it into its task queue. Now assume that **A** sees that task and steals it from **B**'s queue (Line 15). It is now possible for **B** to finish the task it has been working on, reach the barrier, see that **A** has also reached the barrier, and thus exit it and reach the final barrier. In the meantime, **A** exits the barrier (Line 18) and starts working on the new task, potentially generating even more tasks. This can be generalized to more than two threads, and it is easy to see how a load imbalance can happen in this case, with one thread doing all the work while all the other threads are idle at the final barrier.

In order to fix this potential performance issue, we implemented a *peek* function for the queue. Instead of trying and potentially succeeding in dequeuing a task from the task queue *before* exiting the barrier, we first check if there are any tasks available. If yes, then we proceed to first exit the barrier and then try to dequeue one. The reason we chose this over simply exiting the barrier before trying to dequeue a task (without peeking) is because the latter was found to cause extreme scalability issues to the application.

E. Raytrace

Raytrace contains two DCL races on conditional statements in the file *workpool.C*. The synchronization skeleton of *Raytrace* is shown in Figure 4. It essentially extracts work items from a pool, trying to avoid locking the queue if no items are available to be extracted (Line 1 in Fig. 4b). In this example on double-checked locking (DCL), for this 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 store).

We chose to fix the races by simply removing the conditional statements that are executed outside the critical sections, as protecting them with the locks would make little sense.

```
1  GetJob(...) {
2    ...
3    LOCK(gm->wplock);
4    w = gm->workpool[0];
5    if (!w) {
6      gm->wpstat[0] = 0;
7      UNLOCK(gm->wplock);
8      return 0;
9    }
10   gm->workpool[0] =
         w->next;
11   UNLOCK(gm->wplock);
12   ...
13   return 1;
14 }
```

(a)

```
1  if (gm->wpstat[0]==1)
2    if (GetJob(...) == 1)
3      ...
```

(b)

Fig. 4: Implicit synchronization in Raytrace

```
1  Ray_Trace_Non_Adaptively(...) {
2    ...
3    Global->Queue[local_node][0] = 0;
4    while (Global->Queue[num_nodes][0] > 0) {
5      ...
6      ALOCK(Global->QLock, local_node);
7      work = Global->Queue[local_node][0]++;
8      AULOCK(Global->QLock, local_node);
9      while (work < lnum_blocks) {
10       ...
11       ALOCK(Global->QLock, local_node);
12       work = Global->Queue[local_node][0]++;
13       AULOCK(Global->QLock, local_node);
14     }
15     if (my_node == local_node) {
16       ALOCK(Global->QLock, num_nodes);
17       Global->Queue[num_nodes][0]--;
18       AULOCK(Global->QLock, num_nodes);
19     }
20     local_node = (local_node+1)%num_nodes;
21     while (Global->Queue[local_node][0] >= lnum_blocks
           && Global->Queue[num_nodes][0] > 0)
22       local_node = (local_node+1)%num_nodes;
23   }
24 }
```

Fig. 5: Implicit synchronization and a bug in Volrend

F. Ocean-nc

Ocean-nc contains one data race, which happens when computing the input curl of wind stress. Instead of performing the computation, the value is read from another core's memory. The computation itself consists of just one multiplication operation, so in order to fix the race we decided to simply compute the value locally. We know that our modification does not affect the correctness of the application because a) the value is computed the same way in the contiguous version of *Ocean* and b) because it is possible to verify the output results.

Since our modification is trivial, we will not include *Ocean-nc* in any further discussion.

G. Volrend

Volrend exhibits races in its task queues. The queues are implemented as simple integers, using locks for increment and decrement operations (Lines 7, 12, and 17 in Fig. 5), but not for load operations (Lines 4 and 21). The races appear when one thread finishes its task queue and it starts polling the other task queues for work (job stealing). The problem is that both the loop that checks if there is any work to do in

Parameter	Value
Processor frequency	3.0GHz
Block and page size	64 bytes and 4KB
Private L1 cache	32KB, 4-way
L1 cache access time	1 cycle
Shared L2 cache	512KB per bank, 16-way
L2 cache access time	Tag 6 cycles; tag+data 12 cycles
Memory access time	160 cycles
Network topology	2D mesh
Routing technique	Deterministic X-Y
Flit size	16 bytes
Switch-to-switch time	6 cycles

TABLE I: System parameters.

general, as well as the loop that polls every task queue, do not use locks when accessing the queue variables. Of course, the variables in question are set as `volatile`, but that does not guarantee correct ordering. The solution is simple, we just protected the accesses to the task queue with the appropriate locks.

Volrend also contains a bug due to a race condition. Before entering the work loop, each node initializes its own task queue (Line 3). However, it is possible (although unlikely) for a thread **A** to finish all its tasks and then try and steal from another thread **B**, before **B** has initialized its task queue. Fortunately, there is a barrier synchronizing the threads just shortly before this happens, so we fixed the bug by simply moving the queue initialization code before the barrier.

V. Evaluation

We evaluate the behavior of the modified (Splash-3) applications in order to quantify the changes we introduced. Many of the run-time characteristics of the applications depend on the underlying (simulated) hardware, so we do not evaluate every single one of them in depth. Instead, we focus on some of the major performance metrics, namely execution time, network traffic, and scalability. We also present the number of synchronization primitives operations issued during execution, as well as the time spent waiting at them.

This evaluation is intended to give an overview of how our changes affected the applications, and not to investigate if our version is faster or more efficient. Our goal is to produce a correct and properly synchronized version of Splash-2, not a faster one.

A. Simulation Environment

In our evaluation we use Wisconsin GEMS [26], a detailed simulator for multiprocessor systems. We model an in-order processor, that along with the Ruby cycle-accurate memory simulator (provided by GEMS) offers a detailed timing model. The interconnect is modeled with the GARNET network simulator [27]. We instrument the applications using our pintool to obtain information about all memory accesses and synchronization events, similarly to what Monchiero *et al.* [28] propose, and to drive the simulations. As mentioned before, the pintool also ensures that the applications exhibit correct SC behavior during the run. We do this five times for each application, to account for the variability that parallel applications often exhibit [29]. We simulate systems ranging from a single core to 64 cores, implementing a standard directory-based cache coherence protocol (with MESI states) and with the parameters shown in Table I.

We used the `gcc-4.9.0` compiler for compiling all of the applications. This version *will not* remove the backoff function from *Radiosity*. The reason why we opted for an older version instead of the newest one is because we wanted to avoid running into breaking behavior when evaluating the unmodified applications.

We evaluate the Splash-3 suite with the same input parameters as in the Splash-2 paper [1]. Our aim for Splash-3 is to be used for **simulation** with the **same input parameters**, as has been traditionally done. As such, we did not evaluate any alternatives. At the same time, we do not evaluate the whole suite, but only the applications we modified, since characterizing every single application is outside the scope of this paper. For each of the six applications[4] we simulate the entire application, but collect statistics only from start to completion of their parallel section. Some of our results differ from the original Splash-2 characterization paper [1] due to differences in the underlying hardware model.

B. Synchronization

Tables II and III present the average amount of synchronization operations each application uses during its execution, for 8 and 16 threads respectively. The numbers in parentheses indicate the standard deviation, displayed as a percentage of the average.

We start by observing the number of locks. We see that *Barnes*, *FMM*, and *Radiosity* in Splash-3 have one or two orders of magnitude more locks than the ones in Splash-2. That is mostly due to the "benign" data races, which happen often and thus require a lot of lock and unlock operations. This can lead to performance degradation, as it will be discussed in Section V-C.

Next, we have *Cholesky*, *Raytrace*, and *Volrend*. In these applications, the data races are only employed as implicit synchronization, and thus happen much less often. While *Raytrace* and *Volrend* still see an increase in the amount of locking, it is not as significant as in the previous three applications. Furthermore, *Cholesky* runs without any additional lock operations at all.

Unfortunately, in the case of *Radiosity*, we also see an increase in the variability of the number of lock operations. This is not ideal, since it means that an increased number of runs is required in order to achieve more precise results [29]. For the rest of the applications, that is not the case.

For reference, Table II also contains the number of barrier operations. Since we did not add any additional barriers in

[4]We skip Ocean-nc due to the triviality of the changes we introduced.

Benchmark	Lock	Barrier	Signal	Broadcast	Wait
Barnes	34457 (0%)	64 (0%)	0 (0%)	0 (0%)	0 (0%)
Cholesky	14686 (0%)	25 (0%)	0 (0%)	0 (0%)	0 (0%)
FMM	27281 (0%)	160 (0%)	0 (0%)	0 (0%)	0 (0%)
Radiosity	94632 (0%)	73 (1%)	0 (0%)	0 (0%)	0 (0%)
Raytrace	2056 (0%)	1 (0%)	0 (0%)	0 (0%)	0 (0%)
Volrend	63508 (0%)	336 (0%)	0 (0%)	0 (0%)	0 (0%)

Benchmark	Lock	Barrier	Signal	Broadcast	Wait
Barnes	34508 (0%)	128 (0%)	0 (0%)	0 (0%)	0 (0%)
Cholesky	26124 (0%)	54 (5%)	0 (0%)	0 (0%)	0 (0%)
FMM	27807 (0%)	320 (0%)	0 (0%)	0 (0%)	0 (0%)
Radiosity	95729 (0%)	150 (2%)	0 (0%)	0 (0%)	0 (0%)
Raytrace	2064 (0%)	8 (0%)	0 (0%)	0 (0%)	0 (0%)
Volrend	64191 (0%)	672 (0%)	0 (0%)	0 (0%)	0 (0%)

TABLE II: Run-time Synchronization in Splash-2 for 8 (left) and 16 (right) threads (stddev in %).

Benchmark	Lock	Signal	Broadcast	Wait
Barnes	1064030 (0%)	0 (0%)	11833 (0%)	17 (17%)
Cholesky	14686 (0%)	3908 (0%)	0 (0%)	252 (5%)
FMM	252894 (0%)	3759 (0%)	14100 (0%)	184 (1%)
Radiosity	431959 (4%)	0 (0%)	0 (0%)	0 (0%)
Raytrace	2815 (1%)	0 (0%)	0 (0%)	0 (0%)
Volrend	64162 (0%)	0 (0%)	0 (0%)	0 (0%)

Benchmark	Lock	Signal	Broadcast	Wait
Barnes	1064043 (0%)	0 (0%)	11833 (0%)	32 (8%)
Cholesky	26125 (0%)	6762 (0%)	0 (0%)	1197 (3%)
FMM	253611 (0%)	3759 (0%)	14100 (0%)	342 (3%)
Radiosity	1945412 (5%)	0 (0%)	0 (0%)	0 (0%)
Raytrace	3777 (1%)	0 (0%)	0 (0%)	0 (0%)
Volrend	67311 (0%)	0 (0%)	0 (0%)	0 (0%)

TABLE III: Run-time Synchronization in Splash-3 for 8 (left) and 16 (right) threads (stddev in %).

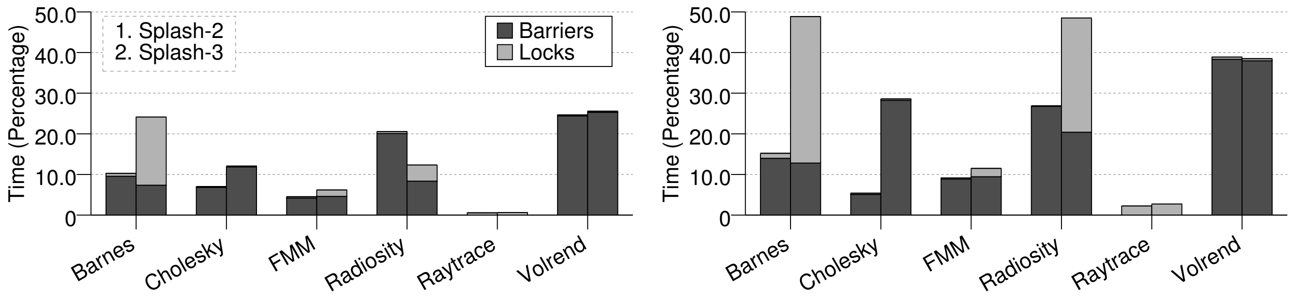

Fig. 6: Percentage of time spent waiting for blocking synchronization primitives for 8 (left) and 16 (right) threads.

the modified applications, there is no need to compare the two versions.

In addition to the number of synchronization operations, we can also observe the time spent waiting on them, seen in Figure 6. We can make two interesting observations here. First of all, in almost all of the cases, the Splash-3 applications spend a bigger percentage of their execution time waiting to acquire locks, which is not surprising since we have introduced more locks. The applications that are affected the less are *Cholesky*, *Raytrace* and *Volrend*, which are also the ones with the smallest amount of locks introduced. Secondly, we can see that (with the exception of *Cholesky* in Splash-2) increasing the number of threads also increases the amount of time spent waiting. This can be explained by the facts that a) there is higher contention and work imbalance, and b) some applications make more synchronization calls.

Finally, in addition to the locks, we also introduced conditional variables in three of the applications, namely *Barnes*, *Cholesky*, and *FMM*. Prior to our modifications, none of the Splash-2 applications actually contained any conditional variables, only *"conditional-variable-like"* synchronization constructs. In that respect, it is interesting to see that the number of signal/broadcast operations in *Barnes* and *FMM* remain the same both for 8 and for 16 threads, while in *Cholesky* doubling the threads leads to a 75% increase in signal operations. The number of wait operations on the other hand vary with the number of threads and also between executions. This was expected, since unlike signal/broadcast, which happen every time data is produced, wait happens only when data happens to *not* be available[5] and is thus very timing sensitive.

C. Performance and Traffic

Performance depends on the underlying hardware (Section V-A), but it is still interesting to see some basic metrics and how our modifications affected the applications.

We start by measuring execution time, and specifically by comparing the execution times for Splash-2 and Splash-3. We see (Fig. 7) that our modifications mostly affected two applications, *Barnes* and *Radiosity*. In the first case, the execution time increased (by 25% for 8 threads, 57% for 16), while in the second, the execution time decreased (by 28% and 58% respectively). The *Radiosity* results were unexpected, since our modifications introduced a large number of additional locks (Section V-B).

In order to figure out why *Radiosity* is significantly faster, we measured the lock contention exhibited in the application. What we noticed is that in the Splash-2 version, the contention is centered around the locks used for the queue. In the Splash-3 version however, the contention is centered around the locks used for the fuzzy barrier. This is due to the facts that a) we have more locks around the barrier variables

[5]In a manner similar to producer-consumer.

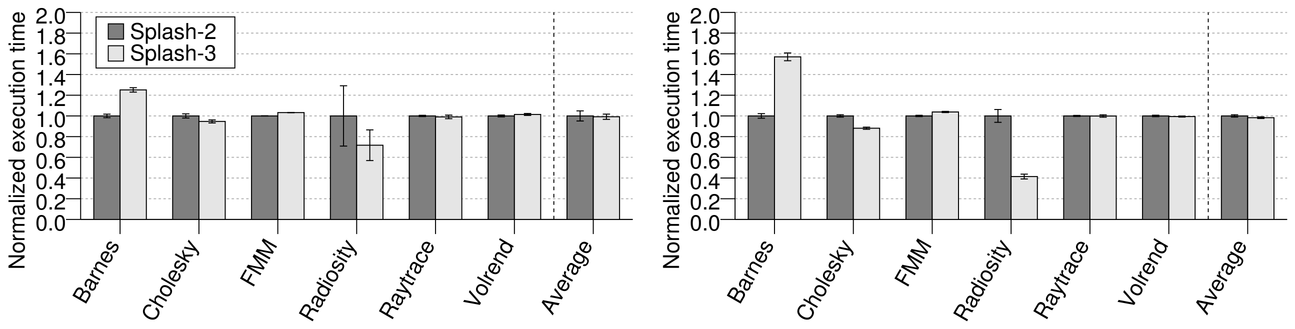

Fig. 7: Normalized execution time for 8 (left) and 16 (right) threads.

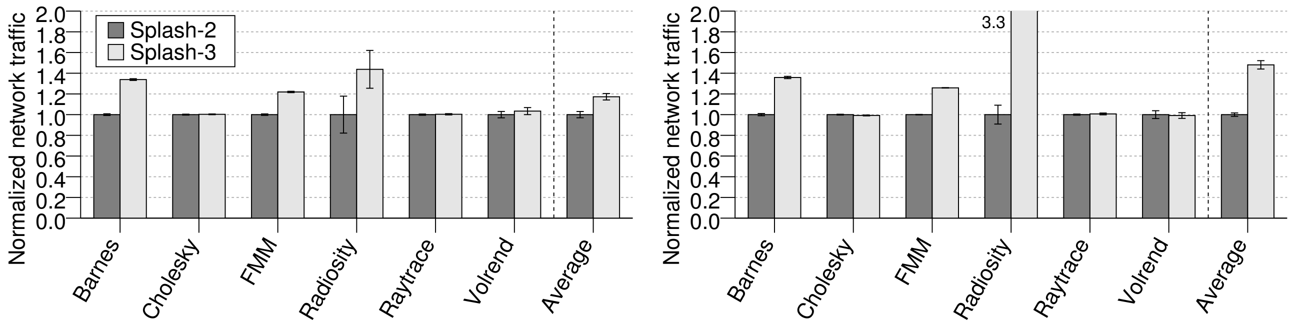

Fig. 8: Normalized network traffic for 8 (left) and 16 (right) threads.

and b) the backoff function is more effective. Moving the contention from the queue to the barrier significantly boosts performance. This is because threads waiting in the fuzzy barrier are not doing any useful work and do not contribute to the forward progress of the application[6] thus slowing them down is inconsequential. In contrast, lock contention in threads that are accessing the queue affects performance.

After *Barnes* and *Radiosity*, *Cholesky*'s execution time is also affected, although not as much (reduced by 5% for 8 threads, 12% for 16). The reason is simple: conditional variables are a more efficient method of synchronization than spin-loops[7]. This becomes more obvious the more threads we are using.

Surprisingly, *FMM*'s execution time increased only insignificantly, even though we introduced a large number of locks there as well. This can be explained by the fact that there is low lock contention, especially in comparison with *Barnes*. At the same time, the data accessed in the critical sections were already shared data, so we have not introduced any significant additional coherence traffic.

Except for the average execution time, its variation is also of interest to us. We see that most applications display at least some small variation, which is expected. Only *Radiosity*'s executions vary significantly, particularly for 8 threads. This can be seen in both the Splash-2 and the Splash-3 versions.

We also measure the network traffic (Fig. 8) for the evaluated applications. As expected, we see significant increases in network traffic for all the applications where we introduced a significant amount of locks. Specifically, *Barnes*'s traffic is increased by 34% and 36% (for 8 and 16 threads respectively), *FMM*'s by 22% and 26%, and *Radiosity*'s by 44% and 228%. Overall, our modifications cause an average increase of network traffic of 17% and 48%, for 8 and 16 threads respectively.

The large increase in traffic observed in *Radiosity* is attributed to the fact that in Splash-2 the threads spend a lot of time spinning while waiting in a barrier, which is performed by accessing the L1 cache and thus generates little traffic, while at the same time making no progress in the execution. The Splash-3 version, on the other hand, executes less instructions, and consequently, causes less accesses to the cache by waiting less time at the barriers. At the same time, the threads cause more cache misses due to frequent data sharing, which indicates that more threads are collaborating actively in the production and consumption of tasks.

Similarly to the execution time, *Radiosity*'s network traffic also varies significantly with each run. Additionally, we see that there is some variation in *Volrend*'s traffic, both for 8 and for 16 threads, in the range of 3% and 4% respectively

[6]Unless they find work to steal in which case they exit the barrier.

[7]Exceptions exist, especially for the cases where the thread(s) only need to wait for a very short period of time.

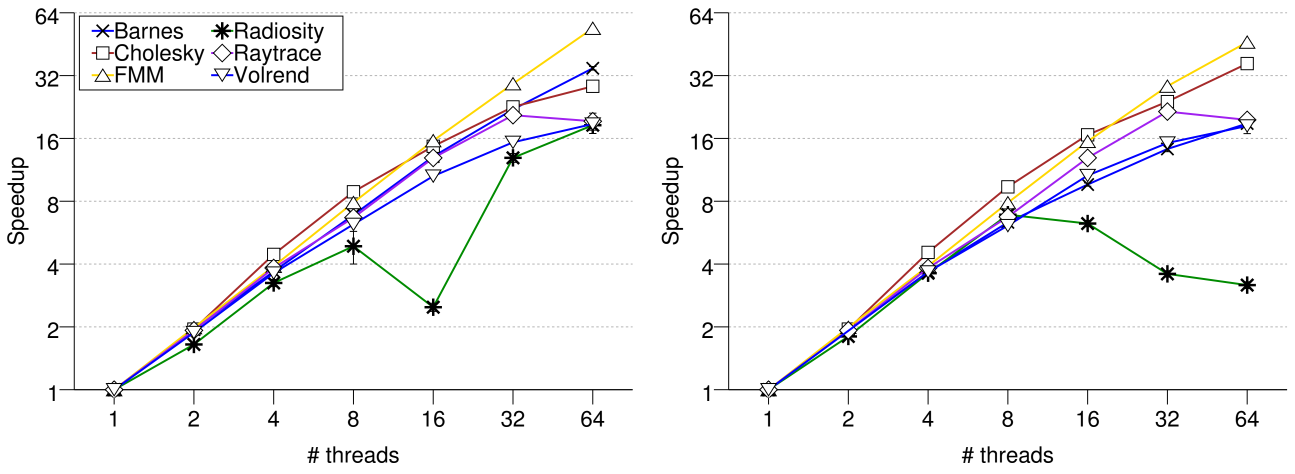

Fig. 9: Scalability of Splash-2 (left) and Splash-3 (right).

for Splash-2 and 3% for both cases for Splash-3.

D. Scalability

Figure 9 presents the scalability (i.e. speedup) of all the applications, from one to 64 threads.

We see that the most scalable application (out of the ones being evaluated) is *FMM*, which comes the closest to perfect scaling with a 56x speedup for the Splash-2 version and 47x for the Splash-3 one (for 64 threads).

Next, we have *Barnes* with 34x and 18x, and *Cholesky* with 28x and 35x respectively. We see that *Barnes* has better scalability in the Splash-2 version, while *Cholesky* is greatly improved in the Splash-3 version. This is explained the same way as the performance differences in those versions: Splash-3 *Barnes* has greater lock contention while *Cholesky* is more efficient with the conditional variables than without.

Then, we have the less scalable applications, *Raytrace* and *Volrend*. *Raytrace* scales up to 32 threads (20x and 21x for Splash-2 and Splash-3 respectively) but its performance degrades after that point. *Volrend* on the other hand does not exhibit a performance degradation, but it only achieves a speedup of 19x for Splash-2 and 18x for Splash-3 (for 64 threads). In addition to this, by observing the curve in the figure, we can clearly see that the scaling is approaching its limit.

Finally, the case of *Radiosity* is a peculiar one. In the Splash-2 version, we observe that for 16 threads there is a sudden performance decrease (only 2.5x speedup), which does not appear for 8 or 32 threads (5x and 13x respectively). The variation in the performance results is too small to allow for the explanation of some unlucky executions. In contrast to the Splash-2 version, our fixes eliminate the inflection point in the speedup curve, so the Splash-3 version does not exhibit this unexpected decrease in performance. However, the application only scales up to 8 threads (7x) before performance starts to drop.

VI. Conclusions

In this paper, we focus on the synchronization of Splash-2. We present data races (and some race conditions) we found in seven Splash-2 benchmarks. We explain not only how those data races limit the researchers in using Splash-2 for different memory models, but also how they are illegal in the current C standard and cause undefined behaviors. We illustrate our point with concrete examples picked from the Splash-2 applications. Finally, we present a version of the suite with the applications corrected, dubbed “Splash-3,” and evaluate how our changes affect the performance characteristics of the applications. Our version is not necessarily “better” than the previous suite, but it is *properly synchronized* and we hope it will aid many researchers in their pursuits.

VII. Future Work

The main issue we encountered during our work with Splash-3, is that a lot of the synchronization is not optimal. This goes for some of both the preexisting and the new synchronization. For example, a large number of locks (particularly the ones used for racy assignments) can be replaced with (potentially *relaxed*, i.e. non-synchronizing atomics that do not follow release/acquire semantics while still not causing data races) atomic memory accesses to the shared data. In addition, some the spinning that could not be replaced with conditional variables can be made more efficient, for example with the addition of proper backoff routines.

Acknowledgments

This work was supported by the Swedish VR-Frame grant 621-2012-5332, and by the ”Fundacin Seneca-Agencia de Ciencia y Tecnologa de la Regin de Murcia” under the project ”Jvenes Lderes en Investigacin” 18956/JLI/13. The simulations were performed on resources provided by SNIC through the Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX).

References

[1] 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)*, Jun. 1995, pp. 24–36.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 72–81.

[3] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. A. Horowitz, and M. S. Lam, "The Stanford DASH multiprocessor," *IEEE Computer*, vol. 25, no. 3, pp. 63–79, Mar. 1992.

[4] G. R. G. Ioannis E. Venetis, "The modified SPLASH-2 home page," http://www.capsl.udel.edu/splash/, Jul. 2007, accessed: 2015-10-09. [Online]. Available: http://www.capsl.udel.edu/splash/

[5] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers (TC)*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[6] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: x86-tso," in *Theorem Proving in Higher Order Logics*. Springer, 2009, pp. 391–407.

[7] ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, December 2011. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853

[8] S. V. Adve and M. D. Hill, "Weak ordering – a new definition," in *17th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1990, pp. 2–14.

[9] ISO, *ISO/IEC 14882:2015 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2015. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=64029

[10] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing c++ concurrency," *SIGPLAN Not.*, vol. 46, no. 1, pp. 55–66, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1925844.1926394

[11] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley Professional, 2014.

[12] A. Ros and S. Kaxiras, "Fast&furious: A tool for detecting covert racing," in *6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA) and 4th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (DITAM)*, Jan. 2015, pp. 1–6.

[13] 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)*, Jun. 1990, pp. 15–26.

[14] *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, ARM, 2014.

[15] *Power ISA, Version 2.07*, IBM, 2013.

[16] S. V. Adve and M. D. Hill, "Sufficient conditions for implementing the data-race-free-1 memory model," University of Wisconsin, Madison, Technical report 1107, Sep. 1992.

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

[18] 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)*, Jun. 2005, pp. 190–200.

[19] 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, Jul. 2004. [Online]. Available: http://www.drdobbs.com/cpp/c-and-the-perils-of-double-checked-locki/184405726

[20] J. Regehr, "Nine ways to break your systems code using volatile," http://blog.regehr.org/archives/28, Feb. 2010, accessed: 2015-10-08. [Online]. Available: http://blog.regehr.org/archives/28

[21] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. ACM*, vol. 17, no. 10, pp. 549–557, Oct. 1974. [Online]. Available: http://doi.acm.org/10.1145/355620.361161

[22] H.-J. Boehm, "How to miscompile programs with benign data races."

[23] D. Vyukov, "Benign data races: what could possibly go wrong?" https://software.intel.com/en-us/blogs/2013/01/06/benign-data-races-what-could-possibly-go-wrong, Jan. 2013, accessed: 2015-10-08. [Online]. Available: https://software.intel.com/en-us/blogs/2013/01/06/benign-data-races-what-could-possibly-go-wrong

[24] H.-J. Boehm, "Threads cannot be implemented as a library," *SIGPLAN Not.*, vol. 40, no. 6, pp. 261–268, Jun. 2005. [Online]. Available: http://doi.acm.org/10.1145/1064978.1065042

[25] H.-J. Boehm and S. V. Adve, "You don't know jack about shared variables or memory models," *Commun. ACM*, vol. 55, no. 2, pp. 48–54, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2076450.2076465

[26] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[27] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[28] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi, "How to simulate 1000 cores," *Computer Architecture News*, vol. 37, no. 2, pp. 10–19, Jul. 2009.

[29] A. R. Alameldeen and D. A. Wood, "Variability in architectural simulations of multi-threaded workloads," in *9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2003, pp. 7–18.