Embedding Semantics of the Single-Producer/Single-Consumer Lock-Free Queue into a Race Detection Tool

Manuel F. Dolz, David del Rio Astorga,
Javier Fernández, J. Daniel García,
Félix García-Carballeira
Department of Computer Science
University Carlos III of Madrid, 28911–Leganés, Spain
drio@pa.uc3m.es
{mdolz,jfmunoz,jdgarcia,fgcarbal}@inf.uc3m.es

Marco Danelutto, Massimo Torquati
Department of Computer Science
University of Pisa, 56127–Pisa, Italy
{marcod,torquati}@di.unipi.it

ABSTRACT

The rapid progress of multi-/many-core architectures has caused data-intensive parallel applications not yet be fully suited for getting the maximum performance. The advent of parallel programming frameworks offering structured patterns has alleviated developers' burden adapting such applications to parallel platforms. For example, the use of synchronization mechanisms in multithreaded applications is essential on shared-cache multi-core architectures. However, ensuring an appropriate use of their interfaces can be challenging, since different memory models plus instruction reordering at compiler/processor levels may influence the occurrence of data races. The benefits of race detectors are formidable in this sense, nevertheless if lock-free data structures with no high-level atomics are used, they may emit false positives. In this paper, we extend the ThreadSanitizer race detection tool in order to support semantics of the general Single-Producer/Single-Consumer (SPSC) lock-free parallel queue and to detect benign data races where it was correctly used. To perform our analysis, we leverage the FastFlow SPSC bounded lock-free queue implementation to test our extensions over a set of μ-benchmarks and real applications on a dual-socket Intel Xeon CPU E5-2695 platform. We demonstrate that this approach can reduce, on average, 30% the number of data race warning messages.

Categories and Subject Descriptors

H.4 [**Multi-/Many-core processors**]: Parallel programming structures; D.1.3 [**Lock-free programming**]: Race detectors

Keywords

Parallel programming; Wait-/lock-free parallel structures; Data race detectors; Semantics

PMAM16 PMAM 2016 The Seventh International Workshop on Programming Models and Applications for Multicores and Manycores

DOI: http://dx.doi.org/10.1145/2883404.2883406

1. INTRODUCTION

As we pave the way towards Exascale computing, the use of multi- and many-core architectures, with possibly one or more co-processors/accelerators, working together to efficiently solve scientific problems becomes a complex challenge that the HPC community needs to face [7, 5]. The adoption of parallel programming frameworks executing multiple processes and/or threads simultaneously, drops developer's burden to design and implement efficient parallel applications from scratch. Despite of this, much of the current software is not yet fully accommodated to run on recent parallel platforms. In most cases, hardware design progresses faster than the parallelization and optimization processes of existing software. To deal with this issue, the use of building blocks implementing core functionalities has been a well accepted approach in the HPC area [9]. Indeed, many of scientific parallel applications leverage efficient parallel kernels from highly-tuned libraries at the bottom of their foodchain. However, these kernels must guarantee correctness and thread-safety in order to generate correct global results.

While parallel programming techniques have been broadly adopted in implementing large scientific applications, concurrency bugs, especially data races, have become more frequent. The adversity in finding data races and deadlocks is a well known problem [8]. Detecting catastrophic errors has been recognized as an arduous task, given that errors may occur only during low-probability sequences of events and may also depend by external factors such as the current machine load. These facts, make data races extremely sensitive in terms of time, the presence of print statements, compiler options, or differences in memory models. Data races are especially difficult to observe, since often they quietly violate data structure invariants rather than cause immediate crashes. Although data race detectors alleviate debugger's task in finding these issues, they are still not perfect [8, 4]. In particular, non-blocking or lock-free structures in which no high-level atomic instructions are used, can still generate false positives, thus blurring developer's vision in finding real data races and making the debugging process harder when tracing back the main cause of the problem.

The contribution of this paper focuses on embedding semantics of the general Single-Producer/Single-Consumer (SPSC) bounded lock-free queue into ThreadSanitizer, a well-known race detector part of the LLVM infrastructure. The formalization and implementation process of the seman-

tics of this specific data structure along with an experimental phase based on a set of applications, demonstrates that it is possible to drop a considerable amount of false positives during the debugging and testing stages of a threaded application. Although the process has focused on the FastFlow SPSC bounded queue, this approach is still valid to any other implementation supporting this data structure.

This paper is organized as follows: Section 3 revisits the main software pieces used in our research. Section 4 describes the semantics of the Single-Producer/Single-Consumer bounded lock-free queue. Section 5 details the implementation of the semantics filtering into the race detection tool. Section 6 performs an evaluation of the reduction of data races using the extended version of ThreadSanitizer over a set of benchmarks. Finally, Section 7 closes the paper with some concluding remarks and future work.

2. RELATED WORK

Available work related to this paper falls into three different categories: *i)* parallel programming frameworks, *ii)* lock-free programming and *iii)* race detection tools.

In the recent years, several projects have demonstrated the benefits of methodologies for effective parallel programming of shared-memory computer architectures. These methodologies have been conceived in the form of programming frameworks in order to provide meaningful programming abstractions. Some examples of frameworks are listed as follows: the OpenMP [22] interface offers a set of compiler directives, library routines and variables that ease the parallelization of applications. On the other hand, OmpSs [10] extends a set of OpenMP-like pragmas to express parallelism at task level together with a runtime in charge of executing them, respecting data-dependencies dictated by a Directed Acyclic Graph (DAG). Other frameworks, such as Intel TBB [25], provide a set C++ templates that embed complications arising from the use of native threading packages. Similarly, Cilk [6] extends the C and C++ languages for task and data parallelism, while FastFlow [2] advocates for a high-level pattern-based parallel interface that supports streaming and data parallelism for heterogeneous and shared-memory platforms.

In the second category we consider lock-free programming, as a non-blocking methodology that bypasses the use of traditional synchronization primitives, such as locks or mutexes, while ensuring thread-safety. The absence of blocking synchronization mechanisms allows increasing performance, since no explicit waiting primitives are needed. However, some constructs may require atomic operations, so that no intermediate states can be seen by another executing threads. Some of the atomic operations used underneath are *test-and-set*, *fetch-and-add*, *compare-and-swap* (CAS) of load-linked/store-conditional (LL/SC) [28]. Basically, these operations atomically combine a *load* and a *store* operation. At hardware level, most of the current architectures from Intel and AMD already implement atomic operations and memory fences [14, 3]. At software level, atomic variables are natively implemented by programming languages (such as C++11) and used by some of the aforementioned frameworks (such as Intel TBB, Cilk or FastFlow) and third-party libraries [27]. Lock-free data structures, such as queues [20], hash-tables [30] and SPSC buffers [12], are typically known to leverage these kind of atomic instructions internally. Lock-free data structures are significantly more complex to implement (consider for example the so called "ABA problem" [23]) and consequently to verify their correctness with respect to lock-based data structures.

In the last category we review a few additional works about race detection tools and techniques. In general, they can be classified into two different groups: static and dynamic. Static algorithms base their results on a static analysis of the application source code using compiler internals, i.e., Abstract Syntax Tree (AST) to analyze dependencies between data structures and synchronization operations [11, 34]. On the contrary, dynamic approaches produce results at run-time using Lamport's happens-before relation, checking for conflicting memory accesses from different threads without any synchronization mechanism [16]. Many examples of tools that fall in this group can be found in the literature [26, 21, 19]. Two consolidated examples of applications that detect errors in multithreaded C and C++ programs using the POSIX threading primitives are DRD [32] and Helgrind [33]. However they are implemented on the top of Valgrind, i.e., driven by a simulator, and thus, making them slower than other approaches. A faster alternative is ThreadSanitizer [29], a race detector that leverages LLVM infrastructure to instrument code at compile-time and detect races at run time, without performing any simulation.

3. BACKGROUND

In this section, we give an overview of the two main software components that have been used to carry out the contribution of this paper. First, we review the FastFlow parallel programming framework, with particular emphasis on the lock-free structures used underneath. Next, we revisit LLVM infrastructure, along with its dynamic analysis tools that identify undefined and suspicious behavior of threads.

3.1 FastFlow and lock-/wait-free structures

The FastFlow parallel programming model is a framework that provides a series of defined, general purpose, customizable and modular parallel patterns conceived as algorithmic skeletons [2]. It pursues the use of high-level, platform-independent structured parallel programming patterns and gives support to develop portable and efficient applications for shared-memory (multi-/many-core, hybrid, GPGPU, FPGA, etc.) and distributed platforms. FastFlow architecture has been designed using a building blocks approach: each layer implements structures built using simpler ones from lower layers. Specifically, its patterns are organized in three main layers: *i)* high level patterns, implementing parallel loops, streams, data-parallel algorithms, workflows of tasks, etc.; *ii)* core patterns, implementing pipelines, farms and feedback structures that allow processes/threads exchanging data among them; and *iii)* building blocks, providing queues, process/thread containers and threads/processes intermediators.

One of the key features of the FastFlow architecture is that threads have, by default, a non-blocking behavior.[1] This, together with lock-less communication channels, allows obtaining high throughput and low latency in shared-memory multi-core architectures. FastFlow communication channels are influenced by FastForward queue [12] and Lam-

[1]If desired, this behavior can be changed in applications that generate long periods of inactivity, e.g., to prevent the CPU from constantly polling, and thus, saving energy.

port's wait-free protocols, thus having the ability to build up streaming networks whose implementation is guaranteed to be correct and efficient through lock-free Single-Producer-Single-Consumer (SPSC) bounded and unbounded queues storing memory references [1].

Different combinations of these SPSC queues can generate more complex streaming networks, e.g., N-to-1, 1-to-M, and N-to-M channels. Although some of these structures require synchronization mechanisms to keep their correctness, FastFlow implements them using helper threads that serialize communications between producers and consumers and avoids the use of expensive synchronization primitives. This results again in wait-free, non-blocking structures that allow to reduce cache coherence overheads.

3.2 The LLVM infrastructure and TSan data race detection tool

The LLVM (Low Level Virtual Machine) is a compiler infrastructure designed to be a set of reusable libraries with well-designed interfaces. Although it was implemented only for C and C++ languages, it currently supports a variety of languages through different front-ends [18]. Furthermore, LLVM generates intermediate code that can be afterwards converted into a machine-dependent assembly code for a specific target platform. It also provides the ability to develop and integrate new modules using the LLVM API in order to perform compile-time analysis and instrumentation.

Taking advantage of the latter feature, various runtime checks and tools have been developed to identify suspicious and undefined behavior of threads. These tools are known as sanitizers and give support for address, memory and thread analysis. Specifically, AddressSanitizer detects the use-after-free and buffer overflows, MemorySanitizer analyzes uninitialized memory reads, and ThreadSanitizer detects data races. Focusing on the goal of our paper, we leverage ThreadSanizier (TSan) [29] to incorporate semantics of a lock-free parallel data structure.

Some of the key features of TSan can be summarized as follows: *i)* it works for applications written in C/C++ or Go, and uses a compile-time instrumentation to check all non-race-free memory access at runtime; *ii)* it leverages detection algorithms to track both lock-sets and the happens-before relations, allowing to switch between the pure happens-before and the hybrid modes; and *iii)* it supports 64-bit architectures using POSIX threads and C++11 threading (within the LLVM `libc++` library) as parallel execution models, being compliant with compiler-built-in atomics and synchronization C++11 primitives.

4. THE LOCK-FREE SINGLE-PRODUCER/-SINGLE-CONSUMER QUEUE

In this section we describe formally the Single-Producer/-Single-Consumer (SPSC) bounded queue and its semantics for the concurrent lock-free version. This allows us to proceed further with our rationale in order to develop semantics and rules for the proper use among entities of the lock-free parallel version.

4.1 Formal definition

Consider a SPSC bounded queue $\mathcal{Q}$ the tuple

$$\mathcal{Q}(buf, pread, pwrite, M),$$

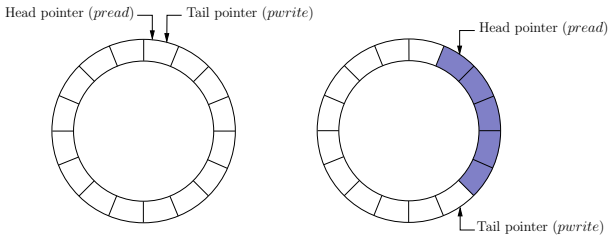

Figure 1: Circular buffer of the Single-Producer/Single-Consumer bounded queue.

where buf is a circular buffer, *pread* plus *pwrite* are internal read and write pointers for the buffer, respectively, and M is a set comprising the following methods:

- `init`: Initializes the buffer buf, allocating space of possibly aligned memory and reseting the internal (*pread* and *pwrite*) pointers by placing them at the beginning of buf. If buf has already been allocated, the method does nothing.
- `reset`: Resets and places the read and write pointers to the beginning of the buffer buf.
- `push`: Enqueues the item into the buffer buf.
- `available`: Returns true if there is at least one room in the buffer buf.
- `pop`: Removes and returns the first item in the buffer buf.
- `empty`: Returns true if the buffer buf is empty.
- `top`: Returns the first item in the buffer buf.
- `buffersize`: Returns the size of the internal queue buffer buf.
- `length`: Returns the number of items currently held by buffer buf.

Figure 1 depicts the internal working of buffer buf from the First-In-First-Out (FIFO) SPSC queue. Initially *pread* and *pwrite* point to the initial position of the buffer, while afterwards some elements have been added at the terminal position through `push` calls and others removed from the head position by means of `pop` calls.

4.2 Semantics of the concurrent lock-free SPSC queue

The implementation of the concurrent SPSC queue for shared cache multi-core systems requires coordination between producer and consumer to ensure proper operation. While different kinds of shared data structures provide different fairness levels requiring diverse synchronization primitives, Lamport in [15, 17] proposes a wait-free Single-Producer/Single-Consumer queue implementation using a circular buffer without requiring any explicit synchronization mechanism. Although Lamport considers a Sequential Consistency memory model, a slightly modified version of this approach is still valid under Total-Store-Order (TSO) and weaker consistency memory models [1]. The importance of these algorithms is that they meet with both wait-free and

lock-free qualities, neglecting the use of blocking constructs, and thus, improving their performance. For our specific case, we leverage the SPSC lock-free bounded queue implementation from the FastFlow programming framework [2], however the methodology presented can be applied to any other implementation featuring this specific data structure.

Nevertheless, the correctness of this specific algorithm is only ensured if several conditions are met. We define these requirements as the following semantics rules:

1. *Fixed roles.* A lock-free concurrent SPSC queue instance can be shared by three different entities if one of them acts as the constructor, the other as the producer and the latter as the consumer; each of them calling only to methods allotted to their specific role. In certain cases, the producer or the consumer can perform the role of the constructor, being only two different entities sharing the same queue. If none of these cases are given, we consider that the queue is misused, thus having an undefined behavior due to potential data races.
2. *Initialization methods.* The constructor can call to methods belonging to:
$$Init = \{\texttt{init}, \texttt{reset}\} : \ Init \subset M.$$
3. *Producer methods.* The producer thread should only invoke a subset of methods in M,
$$Prod = \{\texttt{push}, \texttt{available}\} : \ Prod \subset M.$$
4. *Consumer methods.* The consumer thread should only invoke a subset of methods in M,
$$Cons = \{\texttt{pop}, \texttt{empty}, \texttt{top}\} : \ Cons \subset M.$$
5. *Common methods.* There are methods that can be invoked by both producer and consumer:
$$Comm = \{\texttt{buffersize}, \texttt{length}\} : \ Comm \subset M.$$

Particularly, all subsets allotted to different roles of the queue fulfill $M = Init \cup Prod \cup Cons \cup Comm$. Note also that, methods internally using the *pwrite* pointer are those assigned to the producer, while those using the *pread* pointer are related to the consumer. Methods using none of these pointers or only static parameters, such as the buffer size, can be called by both producer and consumer.

A formalization of the aforementioned semantics is possible if a set C of entity IDs is added as an attribute to each of the preceding subsets of methods in the queue $\mathcal{Q}$. By inserting the ID of the calling entity (or thread) to the corresponding set C of the subsets $Init$, $Prod$ or $Cons$, each time a method belonging to it is invoked, it is possible to control the proper use of the lock-free SPSC queue checking two simple requirements. The first ensures that the cardinality of the C set of the initialization, producer and consumer subsets should always be less or equal than one,

$$|Init.C| \leq 1 \ \wedge \ |Prod.C| \leq 1 \ \wedge \ |Cons.C| \leq 1, \quad (1)$$

hence, only one and the same entity should use methods allotted to its role. The second guarantees that both producer and consumer are performing the right role, i.e.,

$$Prod.C \ \cap \ Cons.C = \varnothing. \quad (2)$$

Note also that all these requirements are valid for the concurrent SPSC queue. In other words, if the producer and consumer entities are different: $|Prod.C \ \cup \ Cons.C| > 1$.

Listing 1 illustrates an execution sequence of a correct use of the lock-free SPSC queue. In this example, 3 threads use the same queue, each of them calling only to its assigned methods, therefore meeting requirements (1) and (2).

Listing 1: Example of execution sequence of a correct use of the SPSC queue.

```
1 Thread 1 call to init      C={1}
2 Thread 1 call to reset     C={1}
3 Thread 2 call to empty     C={2}
4 Thread 2 call to pop       C={2}
5 Thread 3 call to available C={3}
6 Thread 3 call to push      C={3}
```

Listing 2 shows a misuse of the SPSC queue: it violates requirements (1) and (2). The cardinality of set C of methods in $Prod$ and $Cons$ ends up equating to 2 and producer thread with ID 2 calls to methods allotted to the consumer, e.g., $\texttt{push}.C \cap \texttt{pop}.C \neq \emptyset$.

Listing 2: Example of execution sequence of a misuse of the SPSC queue.

```
 1 Thread 1 call to init      C={1}
 2 Thread 1 call to reset     C={1}
 3 Thread 2 call to available C={2}
 4 Thread 2 call to push      C={2}
 5 Thread 3 call to available C={2,3} (Req.1)
 6 Thread 3 call to push      C={2,3} (Req.1)
 7 Thread 4 call to empty     C={4}
 8 Thread 4 call to pop       C={4}
 9 Thread 2 call to empty     C={4,2} (Req.1,2)
10 Thread 2 call to pop       C={4,2} (Req.1,2)
```

5. IMPLEMENTATION

In this section we describe implementation details considered to integrate semantics of the Single-Producer/Single-Consumer parallel data structure into the race detection tool TSan.

To illustrate the workings of TSan, we leverage both execution sequences shown in Listings 1 and 2. Assuming a machine supporting the TSO memory model, both examples would have reported data races in instructions that read and write simultaneously from the same memory address. Listing 4 shows a data race report generated by TSan when executing `push` and `empty` FastFlow functions presented in Listing 3. However, this is only true to some extent. While data races reported for the execution sequence in Listing 2 would have been real due to misuse of the concurrent SPSC queue, those reported in the execution sequence in Listing 1 would have resulted in false positives, as semantics of the concurrent SPSC queue are accomplished at any time. Being the lock-free SPSC queue a thread-safe concurrent data structure, as defined in Section 4, no data races should occur. Therefore, our aim is to endow the race detector with the semantics of this specific data structure in order to filter those "benign" data races and reduce the amount of warnings generated.

Listing 3: FastFlow SPSC bounded lock-free queue producer and consumer implementation.

```
1  /************** PRODUCER FUNCTIONS ****************/
2  bool available() {  return (buf[pwrite] == NULL);  }
3
4  bool push(void* data) {
5     if ( !data ) return false;
6     if ( available() ) {
7        WMB(); // write-memory-barrier
8        buf[pwrite] = data;
9        pwrite += (pwrite+1 >= size) ? (1 - size) : 1;
10       return true;
11    }  return false;
12 }
```

```
13 /************** CONSUMER FUNCTIONS ****************/
14 void* top() const {  return buf[pread];  }
15
16 bool empty() {  return (buf[pread] == NULL);  }
17
18 bool pop(void** data) {
19    if ( !data || empty() ) return false;
20    *data = buf[pread];
21    buf[pread] = NULL;
22    pread += (pread+1 >= size) ? (1-size): 1;
23    return true;
24 }
```

Listing 4: Example of `empty-push` SPSC data race report from TSan. The first two text blocks describe the call stacks of the threads involved in the data race, the third block details the memory location of the data race while the last two give information about the creation of such threads.

```
==================
WARNING: ThreadSanitizer: data race (pid=5181)
Read of size 8 at 0x7d5c0000fc48 by thread T1:
    #0 ff::SWSR_Ptr_Buffer::empty() fastflow-2.0.4/ff/buffer.hpp:186:17 (testSPSC+0x0000004f965a)
    #1 ff::SWSR_Ptr_Buffer::pop(void**) fastflow-2.0.4/ff/buffer.hpp:325 (testSPSC+0x0000004f965a)
    #2 consumer(void*) fastflow-2.0.4/tests/testSPSC.cpp:74:19 (testSPSC+0x0000004f933a)

Previous write of size 8 at 0x7d5c0000fc48 by thread T2:
    #0 ff::SWSR_Ptr_Buffer::push(void*) fastflow-2.0.4/ff/buffer.hpp:239:25 (testSPSC+0x0000004f9527)
    #1 producer(void*) fastflow-2.0.4/tests/testSPSC.cpp:54:19 (testSPSC+0x0000004f91ba)

Location is heap block of size 800 at 0x7d5c0000fc00 allocated by main thread:
    #0 posix_memalign llvm/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:758 (testSPSC+0x000000423967)
    #1 getAlignedMemory(unsigned long, unsigned long) fastflow-2.0.4/ff/sysdep.h:200:9 (testSPSC+0x0000004f9807)
    #2 ff::SWSR_Ptr_Buffer::init(bool) fastflow-2.0.4/ff/buffer.hpp:170 (testSPSC+0x0000004f9807)
    #3 main fastflow-2.0.4/tests/testSPSC.cpp:92:6 (testSPSC+0x0000004f939c)

Thread T1 (tid=5183, running) created by main thread at:
    #0 pthread_create llvm/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:849 (testSPSC+0x000000423ddf)
    #1 main fastflow-2.0.4/tests/testSPSC.cpp:95:11 (testSPSC+0x0000004f93b5)

Thread T2 (tid=5184, finished) created by main thread at:
    #0 pthread_create llvm/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:849 (testSPSC+0x000000423ddf)
    #1 main fastflow-2.0.4/tests/testSPSC.cpp:96:11 (testSPSC+0x0000004f93cc)

SUMMARY: ThreadSanitizer: data race fastflow-2.0.4/ff/buffer.hpp:186:17 in ff::SWSR_Ptr_Buffer::empty()
==================
```

5.1 Supporting semantics of the SPSC queue in TSan

There are several approaches in order to embed SPSC semantics into TSan runtime. A naive but wrong approach to filter "benign" data races is to use the predefined `no_sanitize_thread` TSan attribute in order to avoid instrumentation of routines that have been avowed to generate false data races. This, however, misses real data races given from improper uses of the concurrent SPSC queue. Alternatively, our approach to implement semantics of the concurrent SPSC queue requires the following modifications in TSan runtime internals:

- Given that a multithreaded application can use multiple lock-free SPSC queues simultaneously, with possibly each thread performing different roles in diverse queue instances, it is necessary to univocally identify those in reports obtained by TSan. This can be solved by printing out the C++ implicit `this` pointer associated to the SPSC queue instance involved in the data race report. To do so, it is necessary to design a mechanism to get this pointer from TSan runtime. This is, though, not easy to implement, since the TSan thread responsible for detecting data races is not the same to that incurring the data race, possibly in a different context. There exist several possibilities to implement this approach using communication mechanisms between threads, such as sockets or shared memory. However, all of them require modifications to the user code, something not desirable in our case.

  Our approach follows a different path. We leverage the `libunwind` library [24] to read the stack and base pointers (`sp` and `bp`, respectively) so as to walk backwards on the stack until the parameter is found in the appropriate stack frame.[2] Note that if there are inlined functions within the calling stack, it is not possible to retrieve the desired frame by walking a fixed number of steps back in the stack. Thus, our approach requires Clang `noinline` attribute on inlined user functions and the `-O0` flag to suppress automatic inlining at compile time.

- Afterwards, we implement the semantic rules defined in Section 4. To support this functionality, we leverage a C++ STL `map` container that stores `this` pointers from SPSC queue objects along with `set` structures for each member function of the queue, as key/values pairs. Next, this structure is used to check the semantic requirements of the lock-free SPSC bounded queue, in order to determine whether a data race within a function member of this queue class is benign or not. Finally, we print out only those reports related to real data races.

While in this work we only focused on the semantics of the SPSC queue, the current implementation within TSan

[2]To the best of our knowledge and throughout the experimentation, an object `this` pointer can be found at position `bp` − 1 in the class member function stack frame of a Clang-compiled C++ program running on a x86_64 architecture.

internals could be extended to other lock-free data structures where a formalization of their semantics is possible. We believe this can tremendously aid developers to detect real data races.

6. EXPERIMENTAL RESULTS

After the implementation of the semantic requirements of the SPSC bounded lock-free queue into ThreadSanitizer, we evaluate the detection of data races using a set of μ-benchmarks and real parallel applications from the FastFlow parallel programming framework. In the following we describe the target platform, software and benchmarks sets adopted during our evaluation process.

- ***Target platform***. The analysis and the evaluation has been carried out on a server platform comprised of 2× Intel Xeon Ivy Bridge E5-2695 v2 with a total of 24 cores running at 2.40 GHz, 30 MB of L3 cache and 128 GB of DDR3 RAM, while running Linux Ubuntu 14.04.2 LTS OS on a 3.13.0-57 Linux kernel. We denote this platform as Ivy.

- ***Software***. We leveraged FastFlow v2.0.4 as for the parallel programming framework to test the implementation of the SPSC bounded lock-free queue. As mentioned, for the compilation and race detection process we used the LLVM compiler infrastructure v3.7.0 together with the compiler runtime libraries (`compiler-rt`) including the race detector ThreadSanitizer. Furthermore, we installed the respective LLVM `libc++` and `libc++` ABI in order to support the C++11 standard library for the FastFlow examples.

Featuring the aforementioned hardware and software, we perform our evaluation using the following benchmarks as part of the FastFlow framework. While the first item in the list refers only to the tests included in our μ-benchmarks set, the subsequent application examples are part of the applications set.

- *μ-benchmarks*: They primarily focus to test internal structures and performance of specific FastFlow features. These tests are written in tutorial style and can be effectively used as FastFlow μ-benchmarks. With this, we aim at testing internal workings of FastFlow, therefore testing all possible ways in which a SPSC is used in FastFlow core. We run this applications with the default parameters.

- *Cholesky factorization*: The Cholesky factorization of a symmetric positive definite matrix $A \in \mathbb{R}^{n\times n}$ is given by $A = U^T U$, where the Cholesky factor $U \in \mathbb{R}^{n\times n}$ is upper triangular. Optimization of this algorithm are possible using block-partitioning in conjunction with highly-tuned (BLAS-2 and BLAS-3) kernels, since they reduce the latency associated to the data movement between memory and LLC. During our evaluation, we run both the classic `cholesky` and blocked `cholesky-Block` Cholesky factorizations on a matrix consisting of 20,480×20,480 elements with 40 streams, i.e, a block size of 512×512.

- *Fibonacci*: The Fibonacci numbers is sequence of starting with one or zero and followed by a one, where each subsequent number is the sum of the two previous values, as stated in [13]. Within the FastFlow examples a version of this problem using skeleton programming on a simple stream parallel approach is implemented under `ff_fib`. We set to 100 the length of the Fibonacci series and to 20 the number of streams.

- *Matmul*: This set of applications implement the matrix-matrix multiplication make use of FastFlow parallel pattern structures. The first two tests, `ff_matmul` and `ff_matmmul_v2` use *farms* of tasks each of them computing an element of the output matrix. While the third, `ff_matmul_map`, leverages the *map* construct. All these tests were run using 24 worker threads on matrix sizes of 512×512.

- *Quicksort*: The parallel version of the sorting algorithm Quicksort has also been included in the FastFlow examples. In this case we only run the version of the algorithm based on the farm pattern (`ff_qs`) over an array of 10,000 entries and a threshold of 10.

- *Jacobi*: This algorithm provides the solution of the indefinite Helmholtz equation that is a two-step generalization of the classic Jacobi iteration using complex parameters. It solves the Poisson equation on rectangular grid assuming uniform discretization in each direction and Dirichlet boundary conditions. The FastFlow versions of this application leverage parallel for/reduce and stencil patterns in the `jacobi` and `jacobi_stencil` applications examples, respectively. We set the grid dimensions in both x and y directions to 5,000, the Helmholtz constant to 0.8, the error tolerance to 1.0 and the maximum number of iterations to 1,000.

- *Mandelbrot*: This test is a simple parallel version of the application that computes and displays the Mandelbrot set leveraging FastFlow. Indeed, it is an example of an embarrassingly parallel application. The Mandelbrot set is represented as a matrix of pixels, whose coordinates are orchestrated by a scheduler that dispatches streams to worker threads in a round robin fashion. We run the implementation using FastFlow with and without memory allocator (`mandel_ff` and `mandel_ff_mem_all`, respectively) with a resolution of 640 k-pixel and 1024 maximum number of iterations.

- *n-queens*: The n-queens problem is a generalization of the well-known 8-queens problem supporting a $n \times n$ board-size chess with the purpose of counting all possible solutions. There exist multiple implementations of the n-queens problem using recursive divide and conquer and iterative approaches. We use the fast iterative but parallel FastFlow implementation adapted from the sequential code in [31]. During the evaluation, we executed the both `nq_ff` and `nq_ff_acc` versions computing a board of 21×21 positions.

Note that all the aforementioned benchmarks were compiled using Clang compiler with the following additional flags `-std=c++11`, `-stdlib=libc++`, `-fsanitize=thread`, `-lc++abi` and `-O0`. Afterwards, we executed them using a fixed pool of 24 worker threads, i.e., to fully populate the cores of Ivy. Throughout the execution of these examples, we aim at internally enforcing the use of multiple SPSC

Table 3: Number of SPSC data races caused by pairs of functions for the μ-benchmarks and applications sets.

Benchmark set	SPSC data races		
	push-empty	push-pop	SPSC-other
μ-benchmarks	177	2	4
Applications	60	0	0

queue instances in multiple ways, within different FastFlow parallel constructs. Simultaneously, we collect data races reports generated by TSan for further analysis.

6.1 Analysis of individual SPSC data races

Our first experiment analyzes the frequency of occurrence of SPSC queue-related data races detected by TSan for both μ-benchmarks and application sets. Table 3 combines frequency of the most common function pairs responsible for the data races occurred in both benchmarks tested. As seen, for the μ-benchmark set, a big portion of the data races were caused by the pair of functions `push-empty`. Referring at the code in Listing 3, this data race arose when the producer thread was writing data to the buffer at position *pwrite* (line 8) of the `push` function while the consumer thread was reading from the same position (pointed by *pread*) in `empty` function (line 16). Actually, this was the only type of data race detected for the application set, but not the only for the μ-benchmark set. In this case, we also detected two other pairs of data races. One of them is the tuple `push-pop`, occurring at lines 8 and 20 of the code in Listing 3, also when both threads were writing and reading from the same buffer entry.

Finally, there exist four more occurrences of data races for the μ-benchmarks ("SPSC-other" column) in which only one side involved in the data race was related to a member function of the SPSC queue FastFlow class. Analyzing data race reports we observed that, at some point, a thread executing `posix_memalign` or `malloc` POSIX functions, caused in total 4 possible races when another thread was running `inc`, `pop` or `empty`. This effect was probably caused during the SPSC queue buffer allocation, nevertheless we cannot ensure at this point whether these data races were benign or not. A deep analysis of this section of the code is ongoing.

6.2 Analysis of global and SPSC data races

In this experiment we analyze statistically data races occurred during the execution of both μ-benchmark and application sets with special focus on those related to the SPSC queue.

Figure 2 shows (percentage-wise) the portion of SPSC queue-related data races with respect to the others for both μ-benchmark and application sets. Note that we consider part of the SPSC races those in which only one side was related to a function member of the SPSC queue class. As observed for the μ-benchmark set, roughly 47 % of the data races, on average, were due to SPSC queues. A slightly smaller figure can be appreciated for the application set, decreasing to 34 %. Generally, these percentages give a notion of the importance of this kind of data races occurring in SPSC queue-related functions used to stream data between FastFlow worker threads.

As stated in Section 4, the contribution of the paper is to filter, whenever possible, those "benign" SPSC queue-related data races according to the requirements (1) and (2). Taking advantage of our implementation, in Figure 3 we classify SPSC data races in three different groups:

- *Benign* data races represent those complying both requirements.
- *Undefined* data races stand for those in which TSan failed to restore the stack of one of the threads involved in the data race, and thus, the aforementioned requirements could not be checked.
- *Real* data races stand for those in which, at least, one of the requirements was violated.

Analyzing the percentage breakdown of the different groups, we observe that large percentages of data races (about 50% and 20% for the μ-benchmarks and application set, respectively) were classified as undefined. Since we are not aware of the specific cause that prevented TSan from restoring the stack, we are not confident to classify these data races as benign or real. A deep understanding of the TSan implementation is needed in order to investigate these undefined data races. This further step will be done as future work.

To gain insights into this issue, we performed an extra experiment considering the FastFlow implementation of the bounded and unbounded SPSC queues plus the Lamport version.[3] These tests, `buffer_SPSC`, `buffer_uSPSC` and `buffer_Lamport`, corroborate that percentages of the undefined data races are independent of the queue version. Considering that all implementations are semantically correct but data races are still detected by TSan, we assume that they are all false positives and related to internal issue of TSan.

Similarly, Table 1 combines the breakdown for the different types of data races at SPSC and application levels for both benchmark sets. Additionally, it incorporates figures representing the total number of data races, average of data races per test, and the corresponding percentages over the total data races detected on the application, regardless of their source. Note that the analysis of the SPSC breakdown for the μ-benchmarks and applications sets has already been performed through Figure 3 and 2, respectively. In this table, however, we added a subdivision of data races related to FastFlow, but not to SPSC queues. As observed, this percentage represents approximately a third of total reports.

Finally, the last two columns of Table 1 present figures without and with the data race filtering technique, respectively. As can be seen, we reduce about a third of the number of warnings of data races for both sets tested. Being aware that in this case the filtering technique was only considering the SPSC queue, we are confident that more false positives could have been reduced if semantics for other parallel lock-free data structures would have been considered.

6.3 Analysis of unique data races

To conclude our analysis, we perform a third analysis in which we have filtered redundant data races out from the reports, thus keeping only unique entries. Results for the aforementioned metrics (total, average per test and percentage) are shown in Table 2. Focusing on the SPSC queue

[3]The codes of these structures can be found in the FastFlow SVN repository https://sourceforge.net/p/mc-fastflow/code/HEAD/tree/, more specifically in file `ff/buffer.hpp`.

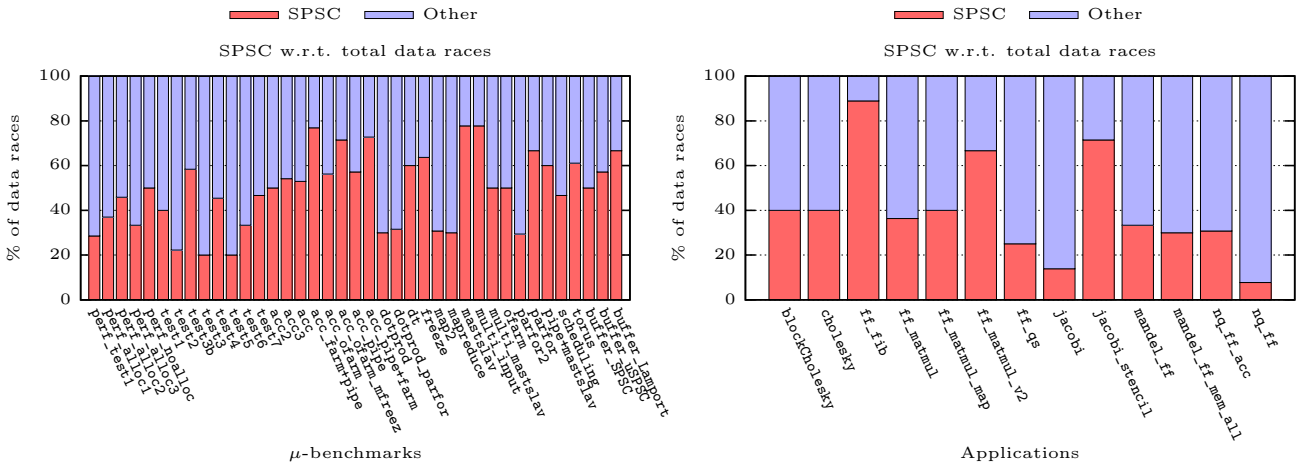

Figure 2: Percentage of SPSC data races with respect to the total for the μ-/benchmarks and applications sets.

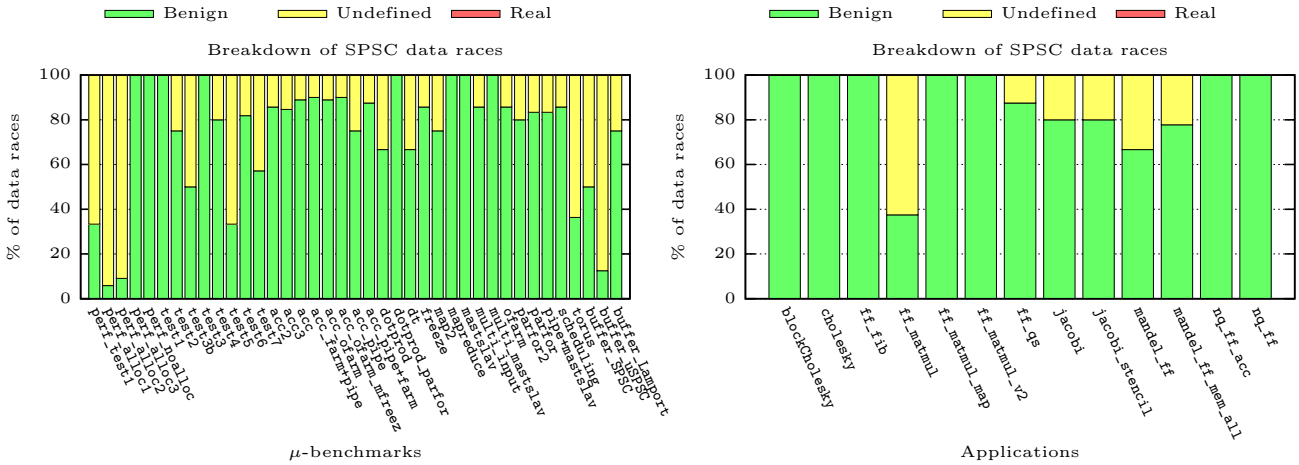

Figure 3: Breakdown of SPSC data races between benign, undefined and real for the μ-benchmarks and applications sets.

Table 1: Statistics of SPSC and application total data races for the μ-benchmarks and applications sets.

Benchmark set	Metrics	SPSC level			Application level			w/o SPSC semantics	w/ SPSC semantics
		Benign	Undefined	Real	SPSC	FastFlow	Others		
μ-benchmarks	Total	187	93	0	280	213	102	595	408
	Per test	4.79	2.38	0.00	7.18	5.46	2.62	15.26	10.46
	Percentage	31.43 %	15.63 %	0.00 %	47.06 %	35.80 %	17.14 %	100.00 %	68.57 %
Applications	Total	60	12	0	72	55	83	210	150
	Per test	4.62	0.92	0.00	5.54	4.23	6.38	16.15	11.54
	Percentage	28.57 %	5.71 %	0.00 %	34.29 %	26.19 %	39.52 %	100.00 %	71.43 %

Table 2: Statistics of SPSC and application unique data races for the μ-benchmarks and applications sets.

Benchmark set	Metrics	SPSC level			Application level			w/o SPSC semantics	w/ SPSC semantics
		Benign	Undefined	Real	SPSC	FastFlow	Others		
μ-benchmarks	Total	72	62	0	134	170	58	362	290
	Per test	1.85	1.59	0.00	3.44	4.36	1.49	9.28	7.44
	Percentage	19.89 %	17.13 %	0.00 %	37.02 %	46.96 %	16.02 %	100.00 %	80.11 %
Applications	Total	19	9	0	28	44	45	117	98
	Per test	1.46	0.69	0.00	2.15	3.38	3.46	9.00	7.54
	Percentage	16.24 %	7.69 %	0.00 %	23.93 %	37.61 %	38.46 %	100.00 %	83.76 %

breakdown, we detect that portions of benign data races has been reduced to approximately 20 % and to 16 % for the μ-benchmark and application sets, respectively. However, percentages for the undefined SPSC data races data races are almost unchanged. Looking at the application level, the percentages of the SPSC data races have dropped accordingly, representing 37 % and 23 % over the total for both benchmark sets. This indicates that almost a 10 % of these data races occurred more than once. Therefore, the fact of dropping SPSC data races, affects positively to reduce noise in debugging traces.

For the data races related to the FastFlow framework, we note that the percentages have increased, being almost a half and a third, for both benchmark sets, respectively. Thus, redundancy of this kind of data races was smaller than that for the SPSC queue (occurring mostly in the same pair of routines, as shown in Figure 3). Other data races remain constant, being again quite heterogeneous and stable with respect to the figures in Table 1.

7. CONCLUSIONS

Data race detectors aid to a great extent developers to easily identify data races in parallel applications. Several post-mortem and dynamical approaches for data race detection have been implemented among a range of tools and plug-ins for compilation infrastructures. However, none of them is aware of the semantics behind the data races. The concurrent use of a shared resource within a correct lock-free parallel structure should not always imply a data race, unless its semantics are violated. In this paper we focused on the general of the Single-Producer/Single-Consumer queue as for the lock-free parallel structure and leveraged, as a use case, the implementation from FastFlow. Specifically, given the building blocks nature of this framework, asynchronous communication channels (SPSC queues) play a very important role, improving throughput and low latency in communicating concurrent threads in a parallel application.

Being aware of the importance of this structure, we formalize the semantics of the SPSC queues and build a set of requirements to determine whether a queue has been properly used or not. Afterwards, we implement the formalization of these semantics into ThreadSanitizer, a well-known dynamic data race detector among the LLVM Clang compiler. The ability of filtering out false positives at runtime, i.e., cases in which the tool detected a potential data race due to a write-read memory access, is a very helpful feature to prevent overwhelming due to warning reports. Also, it allows detecting real data race through a second level of verification semantics. We demonstrate that this technique allows discarding, on average, 66 % and 83 % of the SPSC data races set for the selected benchmark sets, and about 30 % over the total number of data races detected.

Although for the sake of simplicity we have only covered the case of the SPSC bounded queue, in future work we advocate for extending and formalizing semantics of other lock-free parallel data structures. For the case of FastFlow, we plan to consider queues and communication channels build on the top of the SPSC bounded queue, i.e., SPSC unbounded, one-to-many (SPMC), many-to-one (MPSC), and many-to-many (MPMC) queues. Furthermore, we plan to support other memory models (apart from the Total-Store-Order), such as the relaxed one from the IBM Power8 architecture.

8. ACKNOWLEDGMENTS

This work was supported by the EU Project 644235 "RePhrase: Refactoring Parallel Heterogeneous Resource-Aware Applications" under the programme ICT-09-2014.

9. REFERENCES

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. An Efficient Unbounded Lock-Free Queue for Multi-core Systems. In C. Kaklamanis, T. Papatheodorou, and P. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 662–673. Springer Berlin Heidelberg, 2012.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: high-level and efficient streaming on multi-core. In *in Programming Multi-core and Many-core Computing Systems, ser. Parallel and Distributed Computing, S. Pllana*, page 13, 2012.

[3] AMD Corporation. *Revision Guide for AMD Athlon 64 and AMD Opteron Processors*. Number 25759. July 2009.

[4] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[5] S. Ashby and *et al.* The opportunities and challenges of Exascale computing. Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, November 2010.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, Aug. 1995.

[7] S. Borkar. The exascale challenge. In *VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on*, pages 2–3, April 2010.

[8] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM SIGPLAN Notices*, volume 37, pages 211–230. ACM, 2002.

[9] J. Choi, M. Dukhan, X. Liu, and R. Vuduc. Algorithmic time, energy, and power on candidate hpc compute building blocks. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 447–457, May 2014.

[10] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[11] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, Oct. 2003.

[12] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 43–52, New York, NY, USA, 2008. ACM.

[13] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers*. Oxford Science Publications. Clarendon Press, Oxford, 1979.

[14] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US. December 2009.

[15] L. Lamport. Concurrent Reading and Writing. *Commun. ACM*, 20(11):806–811, Nov. 1977.

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

[17] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979.

[18] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[19] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: Effective sampling for lightweight data-race detection. *SIGPLAN Not.*, 44(6):134–143, June 2009.

[20] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[21] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. *ACM SIGPLAN Notices*, 38(10):167–178, 2003.

[22] OpenMP API for parallel programming, version 4.0. http://openmp.org/.

[23] S. Prakash, Y. H. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5):548–559, 1994.

[24] S. Project. The libunwind project. http://www.nongnu.org/libunwind/.

[25] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[26] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[27] B. Schling. *The Boost C++ Libraries*. XML Press, 2011.

[28] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the Cost of Atomic Operations on Modern Architectures. In *24th International Conference on Parallel Architectures and Compilation (PACT'15)*. ACM, Oct. 2015.

[29] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic Race Detection with LLVM Compiler. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 110–114, Berlin, Heidelberg, 2012. Springer-Verlag.

[30] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006.

[31] J. Somers. The N Queens Problem, a study in optimization. http://www.jsomers.com/.

[32] Valgrind-Project. DRD: A Thread Error Detector. http://valgrind.org/docs/manual/drd-manual.html, 2009.

[33] Valgrind-Project. Helgrind: A Data-Race Detector. http://valgrind.org/docs/manual/hg-manual.html, 2009.

[34] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 205–214, New York, NY, USA, 2007. ACM.