The Information Needed for Reproducing Shared Memory Experiments

Vincent Gramoli

Data61-CSIRO and University of Sydney, Australia
vincent.gramoli@sydney.edu.au

Abstract. Reproducibility of experiments is key to research advances. Unfortunately, experiments involving concurrent programs are rarely reproducible. In this paper, we focus on multi-threaded executions where threads synchronize to access shared memory and present a series of causes for performance variations that illustrate the difficulty of reproducing a concurrent experiment. As one can guess, our experimental results are not intended to be reproducible but are meant to illustrate conditions that affect conclusions one can draw out of concurrent experiments.

Keywords: Reproducibility, Synchrobench, artifact, NUMA, cTDP, JIT, pinning

1 Introduction

Science advances faster when researchers do not follow false leads. Interestingly, scientists give the rise of computing tools as a pretext for disclosing information regarding scientific experiments and explains that, with some exceptions, not releasing the source program for results that depend on computation is "intolerable" [1]. The journal *Nature* allows editors to even decide to reject papers if the computer code is unavailable [2]. In computer science research, however, researchers rarely share their source code at the time of publication. Some may not respond to requests asking to share their source code for the sake of reproducibility. And when computer scientists share their code, they sometimes share a different version than what they used in their experiments [3]: "The shoemaker's son always goes barefoot."

That said, remarkable efforts from the programming language community were recently devoted to encourage the reproducibility of computer science experiments [4]. In recent editions of programming language conferences, authors had the opportunity to submit an *artifact* containing their documented source code as well as scripts and required libraries. For example, conferences like OOPSLA and ECOOP accepted artifact submissions since 2013, POPL and PLDI started in 2014 and PPoPP, CGO and CAV started in 2015.[1] After submission, the artifact gets evaluated along four criteria: (i) consistency indicating whether the artifact helps reproducing the results of the paper, (ii) completeness indicating whether the fraction of the reproducible results represent a large fraction of the paper results, (ii) documentation indicating whether the documentation helps applying the method to new inputs, and (iv) simplicity indicating whether the artifact is easy to reuse.

Unfortunately, the replay of shared memory program executions is known to be a difficult problem. One of the reasons is that multi-threaded executions are non-determinstic [5]:

[1] http://evaluate.inf.usi.ch/artifacts.

the output of the program is not tied to its input. This non-determinism affects the debugging process as it makes it difficult to reproduce an error-prone execution that involves a data race at a particular combinaison of points in the executions of multiple threads [6]. It also affects performance monitoring by leading to different performance based on a precise ordering of memory accesses by concurrent threads [7]. Overall, it makes the reproducibility of an experiment highly dependent on a variety of factors external to the program, like the hardware, the operating system, the programming language and the benchmark.

In this paper we show the importance of documenting these environmental factors for reproducibility. To illustrate our claim we measure the performance variations of Synchrobench, a benchmark suite to evaluate synchronization techniques and shared memory programs [8], when playing with OS, language, hardware and benchmark parameters. In particular, we show that core pinning can improve the benchmark performance up to 24%, just-in-time optimizations can lead to 3.4% performance boost, compilation of bytecode to native code can boost performance by 3.9×, and that configurable Termal Design Power (cTDP) can lead to 34% performance boost. Similar to the measurement bias of natural and social sciences [9] our goal is to outline the measurement bias in the particular context of concurrent programming when varying a parameter rather than capturing precisely all the causes of the performance we obtained in a particular case.

In Section 2 we present the problem through a running example. In Section 3 we show the impact of the operating system version on the performance of concurrent programs. In Sections 4 and 5 we show the impact of the hardware configuration and the programming language on the performance, respectively. In Section 6 we show the impact of the definition of benchmark parameters on the performance and Section 7 concludes.

2 The Problem of Insufficient Description

To illustrate the difficulty of reproducing a concurrent program execution, let us take a simple concurrent list-based set benchmark example. The benchmark consists of a linked list data structure implementing a set, exporting *operations* `insert(v)` that returns `false` if `v` belongs to the set, otherwise it inserts the value `v` to the set and returns `true`; `delete(v)` that returns `false` if `v` does not belong to the set, otherwise it removes `v` from the set and returns `true`; and a `contains(v)` that returns `true` if `v` is in the set, otherwise it returns `false`. Indicating that the benchmark written in Java gives the number of operations executed per second with up to k threads on a k-way Intel Xeon machine with 20% updates is insufficient for anyone else to reproduce these experiments. In particular, one must at least indicate details regarding:

- the operating system: the version of the operating system, the memory access policy in use, how threads are pinned to cores;
- the programming language: the version, parameters of the compiler used;
- the hardware: whether overclocking is possible, whether k hardware threads are provided by k independent cores or through hyperthreading;
- the benchmark: how does the benchmark works and what are the parameters.

As we evaluate in the next sections, the impact induced by some changes in each of these environmental settings, meaning the type of operating system, the compiler, the architecture or the benchmark can dramatically affect the performance results. Trying to reproduce such an experiment without such information would likely lead to different results and conclusions.

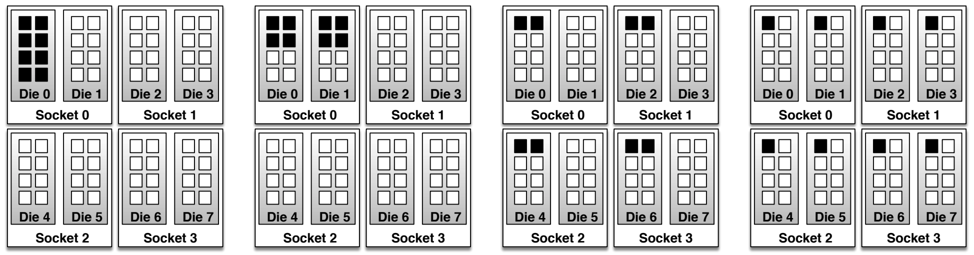
3 The Operating System Impact

The operating system may or may not be aware of multi-threading within cores to decide on an appropriate strategy to pin threads to cores, a strategy that can dramatically impact performance.

3.1 Core pinning

Thread placement or *core pinning* is known to greatly impact performance by either minimizing conflicts or maximizing sharing, typically on TLB and caches. In particular, core pinning can have a higher impact on AMD Opteron than on Sun UltraSPARC [8] and in general the strategy differs depending on the programming language and the operating system used.

To illustrate the impact of core pinning on performance of concurrent programs, we implemented different core pinning strategies in Synchrobench C/C++ on an AMD Opteron 6378 featuring four different sockets. Each AMD socket contains multi-chip modules, meaning that two distinct CPU dies are placed in the same processor package [10]. Each socket embeds two different dies connected to an individual memory controllers—memory controllers are omitted in this figure for the sake of clarity. Each die embeds 8 individual cores, leading to a total of 64 cores as indicated in Figure 1.



(a) SD config: same socket, same die

(b) SND config: same socket, different dies

(c) NSD config: different sockets - same die

(d) NSND config: different sockets - different dies

Fig. 1. Core pinning strategies as 4 block diagrams of the 4 AMD Opteron 6378 multicore machine

Figure 2 depicts the performance of one of the fastest concurrent skip lists to date, the Rotating skip list [11], of the C/C++ version of Synchrobench with four different core pinning strategies. Each experiment ran with C/C++ Synchrobench parameters `-i2M-r4M-t8`, indicating that 8 threads accessed the skip list initialized with 2M elements randomly chosen among a range of 4M elements [8]. The only difference was the way these threads were pinned to cores, as represented in Figure 1. We implemented explicit placement strategies in Synchrobench and measured the performance obtained: SD means same socket and same die first, SND means same socket different dies first, NSD means different sockets and same die first, and NSND means different sockets and different dies first.

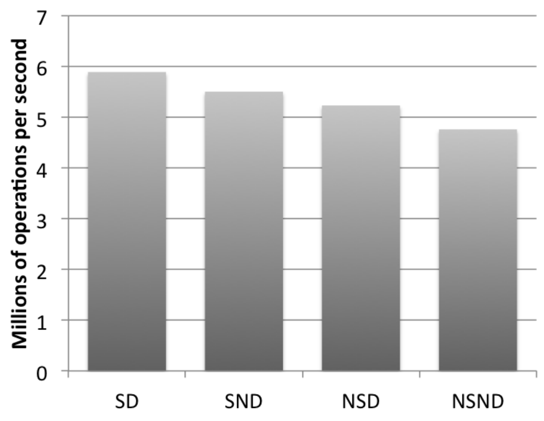


Fig. 2. The impact of the core pinning on a multi-socket NUMA machine: some workloads may benefit from local resource sharing and lead to better performance under a compact core pinning strategy than under a scatter core pinning strategy (error bars represent the sample standard deviation)

The best performance is obtained with the SD configuration while the worst performance is obtained with the NSND configuration. In particular the performance degrades as threads get scattered further apart. This indicates that the performance benefits from the sharing of resources of the same die and the same socket. Overall, we observed that the choice of the configuration could boost the performance by 24%, when comparing the best configuration, SD, to the worst configuration, NSND.

Another important performance factor of concurrent programs executed on multicore machines is the memory [12]. The AMD machine we presented above offers a non-uniform access to memory, hence called NUMA. It is especially important to understand the policy used by the operating system of a NUMA machines to select a memory controller to allocate a page in memory [13].

3.2 Variations across OS versions

The operating system can adopt different core pinning strategies based on its version. This variation implies that the same concurrent program running on two different versions of the same operating system would experience different performances.

A typical example is the Solaris operating system. On Solaris, threads of the HotSpot JVM are bound to lightweight processes. When a lightweight process for a thread is created, the kernel assigns it to a locality group. In versions 10 and 11 of Solaris, the core pinning strategy differs substantially as Dave Dice pointed out in his blog [14]. More precisely, in the version 10 of Solaris, the core pinning strategy balances threads over dies, then over cores, then over pipelines whereas in the version 11 of Solaris, the strategy groups the threads of a unique process on the same locality group until the workload exhausts half of the resources of this locality group. Our previous experience when using Solaris 10 on an UltraSPARC T2 [8] confirmed that the lightweight processes mapped to the HotSpot JVM were scattered across the physical cores.

4 The Hardware Impact

In this section, we illustrate the impact of the hardware on the performance of the concurrent program. In particular, we discuss the difference between exploiting k hardware threads and k cores and the automatic overclocking that may bias conclusions regarding performance scalability with concurrency.

4.1 Core multi-threading

Core multi-threading is a technique used by hardware manufacturers to allow multiple threads to share the pipeline, the CPU and caches and to execute multiple instructions per cycle on a single processor. For example, POWER8 supports simultaneous multi-threading allowing up to 8 hardware contexts to run on a single core. Intel supports hyperthreading allowing up to 2 hardware contexts to run on a single core. With simultaneous multi-threading one physical core appears as multiple processors to the operating system.

In Communications of the ACM [15], we quantified the slowdown due to hyperthreading. We indicated the performance obtained when running some benchmarks on two Xeon machines: one using two single-core hyperthreaded Xeon CPUs and another Xeon with 4 non-hyperthreaded cores. The slow-down was significant at 4 threads as hyperthreading was used in only one of the two machines. Such a difference was explained partially by the fact that in one case, hyperthreading makes two threads share the same processor while in the other case, the processors are not shared. This simple observation led to the conclusion that, in contrast to previous experimental observations, software transactional memories, despite some limitations [16], could scale with the level of concurrency, making it an interesting paradigm rather than a simple "reseach toy".

4.2 Dynamic frequency adjustment

One should be cautious when testing scalability of a concurrent program as its ability to perform better as the level of concurrency increases. The usual scalability graph would plot the performance on the y-axis while the number of threads increases on the x-axis, however, cores may automatically get overclocked if only few threads are active. These higher frequencies at low thread counts can thus result in having fewer threads performing better than more threads. However, this poor scalability is not necessarily due to the contention of the concurrent program, but can be due to an architectural feature as we explain below.

Multicore manufacturers implemented techniques, like dynamic frequency scaling, to reduce the energy consumption [17] when some processes are idle and they also implemented features, like Turbo Boost, that optimizes the performance of one cores when others are inactive. In particular, manufacturers provide *Configurable TDP* (cTDP): Intel explains that the processor may "operate at a power level that is higher than its TDP configuration".[2] The AMD Turbo Core technology increases similarly the core frequency within the thermal and power limits of the accelerated processing unit.[3] This features are

[2] http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

[3] http://www.amd.com/en-us/innovations/software-technologies/turbo-core.

enabled depending on the number of cores running. Similar techniques exist on other architectures as well. The On Chip Controller (OCC) is a co-processor embedded directly on the POWER processor die that controls the frequency, power consumption and temperature to maximize performance while minimizing energy usage [18].

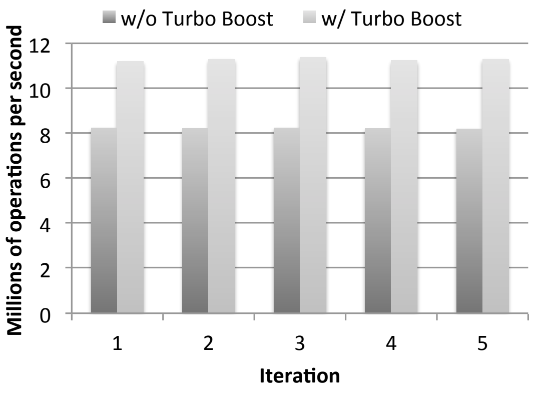


Fig. 3. The impact of the use of dynamic clock frequency adjustment on the performance results: Turbo Boost dynamically increases the clock frequency of a computing core when less cores are active, hence leading potentially to higher single-threaded performance

Figure 3 compares the Synchrobench performance one can obtain with and without Turbo Boost when running the Versioned List benchmark with a single thread (`-t1`) on a machine with two Intel Xeon E5-2450 running 8 hyperthreaded cores each, for a total of 32 hardware threads. More precisely, the list of parameters for Synchrobench Java is `-W0-t1-d5000-u40-i0-r50-bVersionedListSet`. In each of five iterations of the benchmarks, we compare the throughput obtained from the same benchmark with Turbo Boost disabled (w/o Turbo Boost) and with Turbo Boost enabled (w/ Turbo Boost). As its name indicates, Turbo Boost increases performance substantially as we observed a gain in performance between 36% and 38% in each iteration.

5 The Programming Language Impact

The choice of programming languages may affect the performance. For example, Java would favor portability rather than low-level optimizations so that in the JDK 9, the package `sun.misc.Unsafe` would not be usable explicitly. In C/C++, however, one could still pack two data items in one memory word, by exploiting the otherwise unused low-order bit of an x86 aligned memory word. This optimization can speedup the execution by requiring one compare-and-swap to set both data items.

5.1 JVM optimizations

Another optimization may come from running the JVM for long enough. For example, explaining that the plotted value was measured as the average of 5 runs of the experiments may not be enough information to be able to reproduce the same experiment. In particular in Java, repeating the same experiments as part of the same JVM instance may lead to better performance than running it as part of separate JVM instances.

In Synchrobench, one has the option to run a benchmark, like the Versioned List [19], in five consecutive iterations within the same JVM instances: the user simply has to invoke the benchmark with option `-n5` so that the benchmark will run five times in a row, restarting from scratch by cleaning up the data structure between two consecutive runs.

```
java -server -cp bin \
  contention.benchmark.Test -W 0 -t 2 -d 5000 -u 40 -i 0 -r 50 \
  -b linkedlists.lockbased.VersionedListSet -n 5
```

The performance results obtained after running the previous command will be, on average, higher than the performance results obtained after running the following command five times. The only difference is that the user runs the benchmark five times manually, each time specifying that the benchmark should run only once (`-n1`).

```
java -server -cp bin \
  contention.benchmark.Test -W 0 -t 2 -d 5000 -u 40 -i 0 -r 50 \
   -b linkedlists.lockbased.VersionedListSet -n 1
```

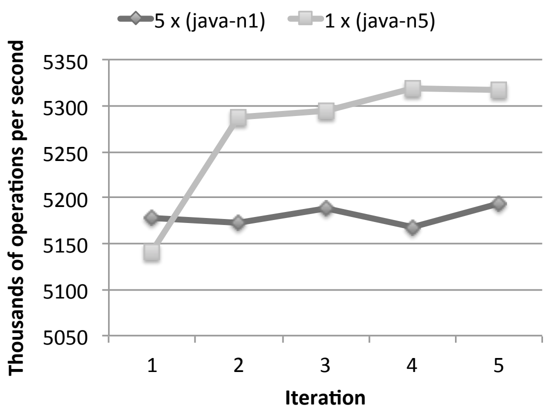


Fig. 4. The impact of the JVM optimizations on performance: the Java code is typically optimized at runtime by the JVM, hence a longer execution within the same JVM instance may lead to better performance than several shorter executions within different instances

Figure 4 shows steady performance results for the five results when each iteration is run as part of individual JVM instances: the variation is up to 4‰ of the minimum throughput.

However, it also shows that the performance varies much more when the five runs are part of the same JVM instance. This variation is up to 3.5% of the minimum throughput. Moreover, the performance obtained in these five consecutive iterations increases with time, which indicates that the performance of the benchmark is optimized during the runtime of the JVM.

Note that in Java, because `synchronized` is built into the JVM it can perform optimizations such as lock elision for thread-confined lock objects and lock-coarsening to eliminate synchronization with intrinsic locks, which indicates that it is better to use `synchronized` locks rather than `ReentrantLock` for performance reasons [20]. There are two Java implementations of the Versioned List [19, 21] in Synchrobench, one uses the `java.util.concurrent.locks.StampedLock` present in the JDK since Java 8, whereas the other uses custom versioned locks. The Versioned list, called `VersionedListSet` used here is the one with the custom versioned locks as opposed to the `VersionedListSetStampLock` benchmark that relies on `StampedLock`.

5.2 Compiler optimizations

For example, running Synchrobench Java with the following parameters:

```
java -Djava.compiler=NONE -server -cp bin \
  contention.benchmark.Test -W 5 -t 4 -d 5000 -u 40 -i 1024  \
  -r 2048 -b linkedlists.lockbased.VersionedListSet
```

will run the Versioned linked list [19] implementing a list-based set initialized with 2^{10} values (-i1024) taken in a range of 2^{11} elements (-r2048), with only 4 threads (-t4), during 5 seconds (-d5000) with attempted update ratio of 40% (-u40). The important parameter `-Djava.compiler=NONE` guarantees that the bytecode will not be compiled to native code during the execution.

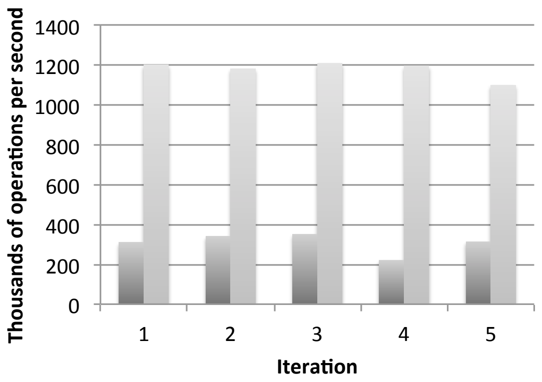


Fig. 5. The impact of the the compilation process on the performance: preventing the Java bytecode from being compiled into native code leads to lower performance than when the code gets compiled

Figure 5 depicts the performance results observed with the compiler enabled and with the compiler disabled while running this workload 5 times on an Intel Xeon with 2 sockets of 8 hyperthreaded cores. The compiler offers a 4-fold speedup on average.

Note that other optimizations exist with different compilers. An example is the GNU compiler collection, gcc, that takes an optimization flag as an argument on the command line to optimize the performance of the program. Hence, forgetting to mention the compilation flags may prevent someone else from reproducing the concurrent experiment. Moreover, recent versions of gcc allow for automatic padding of in-memory structures to minimize automatically false-sharing that may trigger unnecessary cache invalidation leading to performance drops. This is in contrast with earlier versions of the same compiler where padding had to be coded explicitly.

6 Benchmarks

Benchmarks are necessary to demonstrate the performance of new concurrent programs. In particular, macro-benchmarks and applications offer realistic workloads while micro-benchmarks typically offer a refined set of tests to nail down the causes of performance variation. These benchmarks are often tuned to test a particular program and are rarely well documented, making it impossible to reproduce experiments.

6.1 Lack of documentation

Micro-benchmarking is popular to evaluate new concurrent programs. They are invaluable tools that complement macro evaluations and profiling tool boxes. In particular, they are instrumental in confirming how an algorithm can improve the performance of data structures even though the same algorithm negligibly boosts a particular application on a specific hardware or OS [8]. Interestingly, micro-benchmarks are often designed specifically to evaluate a particular concurrent program of synchronization technique [22] and are usually tuned for this purpose. Moreover they are poorly documented, which makes it impossible to reproduce their experiments.

A typical example is the evaluation of contention in concurrent programs. Contention is due to having multiple threads accessing the same shared resources while at least one is trying to modify it. Benchmarks often features a tunable update parameter that allows the programmer to evaluate the performance of a program at different contention levels. In the list-based set example we mention in Section 2, one may think that an update is either an `insert` or a `delete` operation and that the `contains` is clearly not an update but rather a read-only operation. This observation, however, is a bit simplistic as one could also consider that an `insert` and a `delete` that returns `false` without modifying the list-based set are rather read-only operations but are not updates.

6.2 Parameter definitions

To illustrate the importance of precisely defining parameters we compare the performance of two list-based sets using the exact same parameters `-W0-t1-d5000-u40-i0-r50` to compare the performance one would obtain. Update operations in these list algorithms traverse the list until they find the closest node where to insert a node or to delete a node. The Lazy linked list [23] (`linkedlists.lockbased.LazyLinkedListSortedSet`) protects the nodes around this position before the value of the first node is read whereas the Versioned linked list [19] (`linkedlists.lockbased.VersionedListSet`) protects the nodes around this position only if a node is to be inserted or removed.

The problem as indicated on Figure 6 is that whether the update is effective, meaning that it actually modifies the data structure, matters. In case the update is unsuccessful, then the data structure will not be updated and the resulting attempted update can be viewed as a read-only operation.

As the Lazy list locks the data structures even if the structure is not updated, it offers significantly lower performance than the Versioned list. In particular, as the list is initially empty, all the removals executed at the beginning of the experiment will probably fail at removing any node, because the node is likely to be absent. Acquiring locks for these attemtped updates prevents the Lazy list from scaling with the level of concurrency. By constrast, acquiring locks only when necessary allows the Versioned list to scale with the number of hardware threads without suffering from read-only attempted updates. The speedup of the Versioned list over the Lazy list increases with the level of concurrency as well. To conclude, the notion of "update" in benchmarks has to be carefully documented, like it is documented in Synchrobench for example, to indicate whether updates represent modifications or simply invocations of potentially read-only operations.

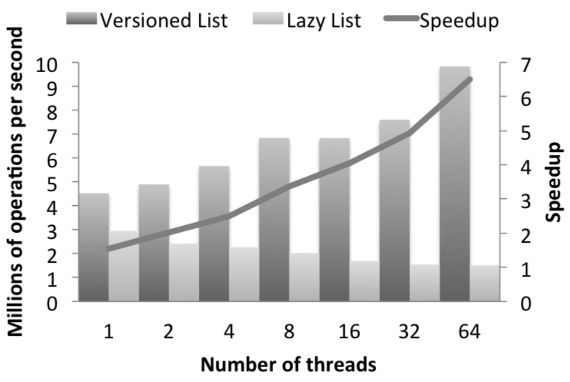


Fig. 6. The impact of attempted vs. effective updates: when update can fail, an algorithm that executes failed updates as read-only operations scales better than an algorithm that acquires locks even during failed updates.

7 Conclusion

The C/C++ and Java versions of Synchrobench were accepted by the artifact evaluation committee of PPoPP 2015. Synchrobench offers an open-source micro-benchmark suite that was used in multiple institutions for teaching and research. In this paper we presented the difficulty of reproducing shared memory experiments by illustrating various factors, not restricted to the benchmark, that may affect the performance of concurrent programs. We encourage researchers to document these factors and make use of existing benchmark suites, like Synchrobench, to simplify reproducibility.

Acknowledgments

Some of the observations reported here were presented at the Winter School organized by the ACM SIGOPS France in March 2016. I wish to thank Tim Harris for fruitful discussions on the topic of publishing experimental results of concurrent programs. This research was supported under Australian Research Council's Discovery Projects funding scheme (project number 160104801) entitled "Data Structures for Multi-Core". Vincent Gramoli is the recipient of the Australian Research Council Discovery International Award.

References

1. Ince, D.C., Hatton, L., Graham-Cumming, J.: The case for open computer programs. Nature **482** (Feb. 2012) 485–488
2. : Code share: Papers in nature journals should make computer code accessible where possible. Nature **514** (Oct. 2014)
3. Collberg, C., Proebsting, T.A.: Repeatability in computer systems research. Commun. ACM **59**(3) (February 2016) 62–69
4. Blackburn, Diwan, Hauswirth, Sweeney, Amaral, Babka, Binder, Brecht, Bulej, Eeckhout, Fischmeister, Frampton, Garner, Georges, Hendren, Hind, Hosking, Jones, Kalibera, Moret, Nystrom, Pankratius, Tuma: Can you trust your experimental results? (2012)
5. Devietti, J., Lucia, B., Ceze, L., Oskin, M.: Dmp: Deterministic shared memory multiprocessing. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XIV (2009) 85–96
6. Russinovich, M., Cogswell, B.: Replay for concurrent non-deterministic shared-memory applications. In: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation. PLDI '96 (1996) 258–266
7. Choi, J.D., Srinivasan, H.: Deterministic replay of java multithreaded applications. In: Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools. SPDT '98 (1998) 48–59
8. Gramoli, V.: More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP 2015 (2015) 1–10
9. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.F.: Producing wrong data without doing anything obviously wrong! In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XIV, New York, NY, USA, ACM (2009) 265–276
10. Braithwaite, R., McCormick, P., chun Feng, W.: Empirical memory-access cost models in multicore numa architectures. In: Proceedings of the International Conference on Parallel Processing (ICPP). (2011)
11. Dick, I., Fekete, A., Gramoli, V.: A skip list for multicore. Concurrency and Computation: Practice and Experience (2016)
12. Drepper, U.: What every programmer should know about memory (2007)
13. Harris, T.: Do not believe everything you read in the papers (Feb. 2016) Personal communication at the NICTA SSRG 4th Summer School.
14. Dice, D.: Thread placement policies on numa systems - update (2012)
15. Dragojević, A., Felber, P., Gramoli, V., Guerraoui, R.: Why STM can be more than a research toy. Commun. ACM **54**(4) (2011) 70–77
16. Gramoli, V., Guerraoui, R.: Democratizing transactional programming. Communications of the ACM (CACM) **57**(1) (Jan 2014) 86–93
17. Groen, M., Gramoli, V.: Multicore vs manycore: The energy cost of concurrency. In: Proceedings of the 22nd International Conference on Parallel and Distributed Computing (Euro-Par). (2016)
18. Rosendahl, T.: On chip controller (occ). In: First Annual OpenPOWER Summit. (2015)
19. Gramoli, V., Kuznetsov, P., Ravi, S., Shang, D.: A concurrency-optimal list-based set. Technical Report 1502.01633v1, arxiv (Feb 2015)

20. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Lea, D., Holmes, D.: Java Concurrency in Practice. Addison-Wesley Professional (2005)
21. Gramoli, V., Kuznetsov, P., Ravi, S., Shang, D.: Brief announcement: A concurrency-optimal list-based set. In: 29th International Symposium on Distributed Computing (DISC). (2015)
22. Harmanci, D., Felber, P., Gramoli, V., Fetzer, C.: Tmunit: Testing transactional memories. In: 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT). (2009)
23. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. In: Proceedings of the 9th International Conference on Principles of Distributed Systems. OPODIS'05 (2006) 3–16