Continuously Measuring Critical Section Pressure with the Free-Lunch Profiler

Florian David*† Gaël Thomas*† Julia Lawall†* Gilles Muller†*

*Sorbonne Universités, UPMC, LIP6 †Inria, Whisper team

firsname.lastname@lip6.fr

Abstract

Today, Java is regularly used to implement large multi-threaded server-class applications that use locks to protect access to shared data. However, understanding the impact of locks on the performance of a system is complex, and thus the use of locks can impede the progress of threads on configurations that were not anticipated by the developer, during specific phases of the execution. In this paper, we propose Free Lunch, a new lock profiler for Java application servers, specifically designed to identify, *in-vivo*, phases where the progress of the threads is impeded by a lock. Free Lunch is designed around a new metric, *critical section pressure* (CSP), which directly correlates the progress of the threads to each of the locks. Using Free Lunch, we have identified phases of high CSP, which were hidden with other lock profilers, in the distributed Cassandra NoSQL database and in several applications from the DaCapo 9.12, the SPECjvm-2008 and the SPECjbb2005 benchmark suites. Our evaluation of Free Lunch shows that its overhead is never greater than 6%, making it suitable for *in-vivo* use.

Categories and Subject Descriptors D.2.8 [*Software engineering*]: Metrics – Performance measures

Keywords Performance Analysis, Locks, Multicore, Java

1. Introduction

Today, Java is regularly used to implement complex multi-threaded server-class applications such as databases [15, 26] and web servers [3, 21], where responsiveness and throughput are critical for a good user experience. Such server applications are designed around the use of shared data, that are accessed within critical sections, protected by locks, to ensure consistency. However, because of the complexity of these servers, some critical sections may not be efficient in all execution configurations. Such critical sections can impede the progress of many threads in specific settings, which can drastically degrade the time for the server to process requests. Therefore, throughput and responsiveness may be hampered in situations that are difficult to simulate exhaustively or that only arise on specific architectures that are not available to the developer. As there is no technique to statically identify such critical sections, there is a need for an *in-vivo* profiler that continuously monitors the application while it is running. Current lock profilers for the widely used state-of-the-art Hotspot Java virtual machine, however, incur a substantial overhead, making their use only acceptable in an *in-vitro* development setting.

Additionally, effective profiling of Java server-class applications requires the use of a metric that reports the slowdown of the server caused by a lock and that takes into account the fact that server-class applications have long running times with various execution phases. Existing Java lock profilers report on the average contention for each lock over the entire application execution in terms of a variety of metrics. These metrics, however, focus on identifying the most used or contended locks, but do not correlate the results to the progress of the threads, which makes them unable to indicate whether an identified lock is a bottleneck. For example, on a classical synchronization pattern such as a fork-join, we have observed that a frequently used or contended lock does not necessarily impede thread progress. Furthermore, by reporting only an average over the entire application execution, these lock profilers are not able to identify local variations due to the properties of the different phases of the application. Localized contention within a single phase may harm responsiveness, but be masked in the profiling results by a long overall execution time.

These issues are illustrated by a problem that was reported two years ago in version 1.0.0 of the distributed NoSQL database Cassandra [26].[1] Under a specific setting, with three Cassandra nodes and a replication factor of three, when a node crashes, the latency of Cassandra is multi-

OOPSLA '14, October 20–24, 2014, Portland, Oregon, US.

http://dx.doi.org/10.1145/2660193.2660210

[1] `https://issues.apache.org/jira/browse/CASSANDRA-3386`.

plied by twenty. This slowdown is caused by a lock used in the implementation of *hinted handoff*,[2] by which live nodes record their transactions for the purpose of later replay by the crashed node. The original developers seem not to have tested this specific scenario, or they tested it but were not able to cause the problem. Moreover, even if the scenario was by chance executed, current profilers would be unable to identify the cause of the bottleneck if the scenario was activated during a long run that hides the contention phase.

To address these issues, there is a need for a profiler with the following properties: i) the profiler must incur little overhead in order to be used *in-vivo*, ii) the profiler must use a metric that indicates whether a lock impedes thread progress, and iii) the profiler should recompute this metric periodically, to be sensitive to the different phases of the application.

In this paper, we propose a new lock profiler, called Free Lunch, designed around a new contention metric, *critical section pressure* (CSP). Free Lunch is especially targeted towards identifying *phases* of high CSP *in-vivo*. We define the CSP as the percentage of time spent by the application threads in acquiring the lock during a time interval, which directly indicates the percentage of time where threads are unable to make progress, and thus the loss in performance. When the CSP of a lock reaches a threshold, Free Lunch reports the identity of the lock back to developers, along with a call stack reaching a critical section protected by the incriminated lock, just as applications and operating systems now commonly report back to developers about crashes and other unexpected situations [13].

In order to make *in-vivo* profiling acceptable, Free Lunch must incur little overhead. To reduce the overhead, Free Lunch leverages the internal lock structures of the Java Virtual Machine (JVM). These structures are already thread-safe and thus Free Lunch does not require any additional synchronization to store the profiling data. Free Lunch also injects the process of periodically computing the CSP into the JVM's existing periodic lock management operations in order to avoid extra inspections of threads or monitors. As a result, Free Lunch only adds eleven instructions to the lock acquiring function on an amd64 architecture.

We have implemented Free Lunch in the Hotspot 7 JVM. This implementation only modifies 420 lines of code, mainly in the locking subsystem, suggesting that it should be easy to implement in another JVM. We compare Free Lunch with other profilers on a 48-core AMD Magny-Cours machine in terms of both the performance penalty and the usefulness of the profiling results. Our results are as follows:

- Theoretically and experimentally, we have found that the lock contention metrics used by the existing Java lock profilers MSDK [32], Java Lock Monitor [30], Java Lock Analyser [20], IBM Health Center [16], HPROF [17], MSDK [32], JProfiler [23] and Yourkit [41] are inappropriate to identify whether a lock impedes thread progress.
- Free Lunch makes it possible to detect a previously unreported phase with a high CSP in the log replay subsystem of Cassandra. This issue is triggered under a specific scenario and only during a phase of the run, which makes it difficult to detect with current profilers.
- Free Lunch makes it possible to identify six locks with high CSP in six standard benchmark applications. Based on these results, we have improved the performance of one of these applications (Xalan) by 15% by changing only a single line of code. As the lock is only contended during half of the run, all other profilers largely underestimate its impact on performance. For the other applications, the information returned by Free Lunch helped us verify that the locking behavior could not easily be improved.
- On the DaCapo 9.12 benchmark suite [5], the SPECjvm-2008 benchmark suite [37] and the SPECjbb2005 benchmark [36], we find that there is no application for which the average performance overhead of Free Lunch is greater than 6%. This result shows that a CSP profiler can have an acceptable performance impact for *in-vivo* profiling.
- The lock profilers compatible with Hotspot, HPROF [17], JProfiler [23], Yourkit [41] and MSDK [32], on the same set of benchmarks incur a performance overhead of up to 4 times, 7 times, 1980 times and 42 times, respectively, making them unacceptable for *in-vivo* profiling.

The rest of this paper is organized as follows. Section 2 presents how synchronization is implemented in JVMs and a study of the metrics used in the state-of-the-art in Java lock profiler. Section 3 presents the design of Free Lunch and Section 4 presents its implementation. We compare the overhead of Free Lunch to that of existing profilers in Section 5 and assess the value of the information produced by Free Lunch in Section 6. Section 7 presents related work and Section 8 concludes.

2. Background

In this section, we first describe the implementation of synchronization in Hotspot 7. The same implementation strategy is used in other modern JVMs, such as Jikes RVM [1] and VMKit [12]. Free Lunch leverages this implementation to perform profiling efficiently. We then present the seven state-of-the-art Java lock profilers of which we are aware and discuss the limitations of their metrics in the context of Java server profiling.

2.1 Locking in the Hotspot 7 JVM

In Java, each object has an associated *monitor* [14], which is comprised of a lock and a condition variable. As typically

[2] `http://wiki.apache.org/cassandra/HintedHandoff`

only a few Java objects are actually used for synchronization, Hotspot 7 includes an optimization that minimizes the monitors' memory consumption [4]. This optimization is based on the following observations: i) when no thread is blocked while waiting for the lock of the monitor, there is no need for a queue of blocked threads, and ii) when no thread is waiting on the condition variable of the monitor, there is no need for a queue of waiting threads.

When both conditions hold, the monitor is considered to be not contended and the Hotspot 7 JVM can represent the monitor in *flat mode* (see Figure 1). In this case, the Hotspot 7 JVM stores a compact representation of the monitor directly in the Java object header. The monitor becomes contended when a thread tries to acquire the monitor lock while it is already held by another thread or when a thread starts waiting on the monitor condition variable. In both cases, the Hotspot 7 JVM *inflates* the monitor, so that it contains thread queues. The Java header then references the inflated monitor structure, which contains the data of the original header.

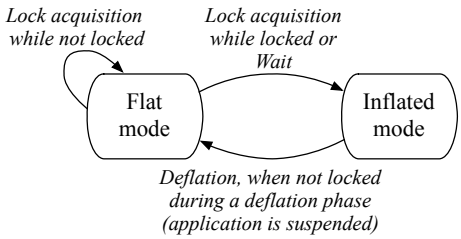


Figure 1. Transitions between flat and inflated mode.

If an inflated monitor becomes not contended because it has no waiting threads for both the lock or the condition variable, the Hotspot 7 JVM eventually *deflates* the monitor into flat mode. To prevent concurrent accesses from the application to the inflated monitor structure, the Hotspot 7 JVM deflates a monitor only when the application is suspended. Hotspot 7 exploits the fact that it already regularly suspends all the threads in order to collect memory, deoptimize the code of a method or redefine a class [35]. Hotspot 7 leverages this synchronization to perform a deflation cycle each time all threads are suspended. During a deflation cycle, Hotspot 7 inspects all the inflated monitors. If a monitor is not contended, Hotspot 7 deflates it into flat mode.

2.2 Lock contention metrics

In this section, we study the metrics used by the seven state-of-the-art profilers of which we are aware. Four of them are designed for the Hotspot 7 JVM (HPROF [17], Yourkit [41], JProfiler [23] and MSDK [32]) and three for the IBM J9 JVM, (JLM from the Performance Inspector suite [30], JLA [20], and Health Center [16]). These profilers focus on ordering the locks, from the most contended to the least contended, using a variety of metrics. However, none of these metrics are correlated to the progress of threads, and thus they do not indicate which locks actually hamper responsiveness in the context of a server.

In the rest of this section, we illustrate this limitation using two classical scenarios for synchronizing threads, (generalized) ping-pong and fork-join, which idealize typical execution patterns performed by servers. We demonstrate that each of the metrics is unable to report whether a lock impedes thread progress for at least one of the scenarios.

2.2.1 Synchronization scenarios

The *generalized ping-pong scenario*, presented in Figure 2, models a server with different kinds of threads, that execute different parts of the server. Two threads, called the ping-pong threads, execute an infinite loop in mutual exclusion. On each loop iteration, a ping-pong thread acquires a lock, performs some processing, and releases the lock. The remaining threads do not take the lock. For example, the two ping-pong threads could take charge of the writes of dirty objects to persistent storage, while the other threads take charge of the logic of the server. In this generalized ping-pong scenario, the progress of the two threads running in mutual exclusion is severely impacted by the lock, such that at any given time only one of them can run. On the other hand, the lock does not impede the progress of the other threads, and overall, the lock does not impede the progress of the application if it uses many other threads. In order to assess if thread progress of the server is impeded by the lock, we would thus like the value of a lock profiling metric to decrease as the number of threads increases.

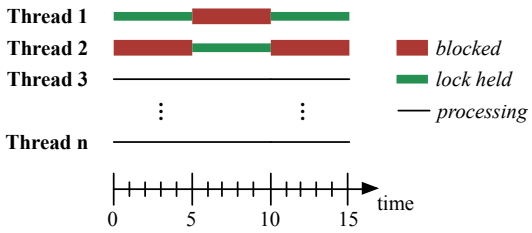


Figure 2. A generalized ping-pong scenario.

In the *fork-join scenario* shown in Figure 3, a master thread distributes work to worker threads and waits for the result. The scenario involves the monitor methods `wait()`, which waits on a condition variable, `notifyAll()`, which wakes all threads waiting on a condition variable, and `notify()`, which wakes a single thread waiting on a condition variable. The three methods must be called with the monitor lock held. The `wait()` method additionally releases the lock before suspending the thread, and reacquires the lock when the thread is reawakened.

The workers alternate between performing processing in parallel (narrow solid lines) and waiting to be awakened by the master (red and green thick lines and dashed lines). While the Java specification does not define an order in

which threads waiting on a condition variable are awakened, to simplify our analysis, we assume that threads are awakened in FIFO order, meaning that `notify()` wakes the thread that has waited the longest on the condition variable. We also suppose that the processing phase takes the same time for each worker.

Initially, the master holds the lock and the workers are waiting, having previously invoked the `wait()` method. At time 0, the master wakes the workers using `notifyAll()`. Each worker receives the notification at time 1. According to the semantics of `wait()`, each worker then has to reacquire the lock before continuing. Thus, all workers block at time 1 while waiting for the master to release the lock. At time 2, the master releases the lock by invoking `wait()`. This leads to a cascade of lock acquisitions among the workers, at times 2-5, with each worker holding the lock for only one time unit. The workers then perform their processing in parallel. When each worker completes its processing, it again enters the critical section, at times 8, 9, 10, and 11, respectively, to be able to invoke `wait()` (times 9-14), to wait for the master. This entails acquiring the lock, and then releasing it via the `wait()` method. Finally, when the fourth worker completes its processing (time 11), it acquires the lock and uses `notify()` to wake the master (time 12). At time 13, the master must reacquire the lock, which is currently held by the fourth worker. The fourth worker releases the lock when it invokes `wait()` (time 14), unblocking the master and starting the entire scenario again.

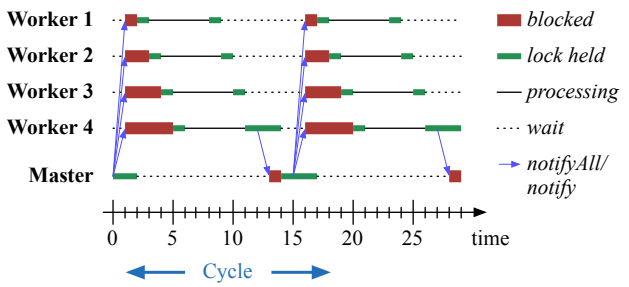


Figure 3. A fork-join scenario.

In this scenario, all of the workers are repeatedly blocked on the same lock, while trying to reacquire the lock in the `wait()` method. If the processing time of the workers is small, the workers are unable to progress during long periods of time as compared to the time of a cycle, while it is the opposite if the processing time is large. A metric should reflect this trade-off.

2.2.2 Analysis of the metrics

We now analyse the metrics proposed by the seven profilers on the two synchronization scenarios. Our analysis focuses on the ability of the metric to indicate whether the threads are unable to progress, as our primary concern is to identify whether a lock hampers the responsiveness of a server. Table 1 presents the metrics and summarizes our analysis. Overall, we see that although some tools provide metrics that report values that scale with the impact on thread progress in some scenarios, in each case there is at least one scenario on which the result does not indicate thread progress, and the user has no way to know which metric value to take into account. In Section 5, we confirm this analysis using experiments.

Metrics based on the number of failed acquisitions. Several profilers propose metrics based on the number of failed lock acquisitions, i.e., the number of times when the lock acquisition method detects that the lock is already held. The idea behind these metrics is that the probability of a lock acquisition failing increases with contention.

JLM, JLA and Health Center report the number of failed acquisitions divided by the total number of acquisitions. With the fork-join scenario (see Figure 3), the result is 5/9, with 4 failed acquisitions by the workers at time 2, 4 successful acquisitions by the workers at times 8, 9, 10 and 11, respectively, and 1 failed acquisition by the master at time 13. This result is a constant and does not reflect that the synchronization only impedes thread progress when the processing time is small.

The same profilers also report the absolute number of failed acquisitions. From this information, we can deduce the rate of failed acquisitions over time by dividing it by the time elapsed during the application run. For the generalized ping-pong scenario, after each round of processing, which takes place with the lock held, the two ping-pong threads are trying to acquire the lock and one of them will fail. The number of fails per time unit is thus equal to one divided by the time of the processing function (the narrow green rectangle in Figure 2). The number of fails per time unit is thus not related to the number of threads, but to the processing time. It is thus inadequate to indicate whether threads are unable to progress.

Thus, the number of failed acquisitions does not seem to indicate whether many threads are blocked by the lock. It is useful to understand which lock is the most contended, but a highly contended lock does not inevitably impede thread progress.

Metrics based on the critical section time. Other widely used metrics are based on the time spent by the application in the critical sections associated with a lock. The idea behind these metrics is that if a lock is a bottleneck, an application will spend most of its time in critical sections.

JLM, JLA and Health Center use this metric as well. They report the amount of time spent by the application in the critical sections associated to a given lock divided by the number of acquisitions of that lock, *i.e.*, the average critical section time. On the generalized ping-pong scenario (see Figure 2), regardless of the number of threads, the average time spent in critical sections (the duration of the green thick

Table 1. Ability of the metrics to assess the thread progress.

Contention metric	Scenario		Profilers
	ping-pong	fork-join	
# failed acquisitions / # of acquisitions	+	-	JLM, JLA, Health Center
# failed acquisitions / Elapsed time	-	-	JLM, JLA, MSDK, Health Center
Total CS time of a lock / # of acquisitions	-	+	JLM, JLA, MSDK, Health Center
Acquiring time of a lock / Acquiring time of all locks	-	-	HPROF
Acquiring time of a lock / Elapsed time	-	-	HPROF, JProfiler, Yourkit

line) remains the same. The metric is thus unable to indicate whether the lock impedes thread progress.

We conclude that the time spent in critical sections does not necessarily indicate whether many threads are blocked. It is only useful to understand which critical sections take the longest time, but long critical sections do not necessarily impede thread progress.

Metrics based on the acquiring time. HPROF, JProfiler and Yourkit report the time spent by the application in acquiring each lock. During the acquiring time, threads are unable to execute, which makes acquiring time an interesting indicator of thread progress.

To provide a meaningful measure of thread progress, the acquiring time has to be related to the overall execution time of the threads. However, JProfiler and Yourkit only report the elapsed time of the application (difference between the start time and the end time), which does not take into account the execution times of the individual threads. Without knowing the number of threads, which can evolve during the execution, it is not possible to determine whether the lock is a bottleneck. For example, on the generalized ping-pong scenario, the metric indicates that 100% of the elapsed time is spent in acquiring the lock (large red lines), regardless of the number of threads.

HPROF also reports the acquiring time of each lock divided by the total time spent by the application in acquiring any lock. This metric is useful to identify the most problematic locks, but is unable to indicate whether a lock actually impedes thread progress. In the ping-pong scenario, for example, the metric again indicates that 100% of the acquiring time is spent in the only lock. The metric is thus not related to the number of threads and is unable to identify whether the lock impede the threads' progress.

3. Free Lunch Design

The goal of Free Lunch is to identify the locks that most impede thread progress, and to regularly measure the impact of locks on thread progress over time. We now describe our design decisions with respect to the definition of our contention metric, the duration of the measurement interval, the information that Free Lunch reports to the developer, and the limitations of our design.

3.1 Free Lunch metric

In designing a metric that can reflect thread progress, we first observe that a thread is unable to progress while it blocks during a lock acquisition. However, taking into account only this acquiring time is not sufficient: we have seen that HPROF, YourKit and JProfiler also use the acquiring time, but the resulting metrics are unable to indicate if the lock actually impedes thread progress (see Table 1). Our proposal is to relate the acquiring time to the accumulated running time of the threads by defining the *CSP of a lock* as the ratio of i) the time spent by the threads in acquiring the lock and ii) the cumulated running time of these threads.

To make this definition precise, we need to define the running time and the acquiring time of a thread, considering, in particular, how to account for cases where the thread is blocked or scheduled out for various reasons. Specifically, we exclude from the running time the time where a thread waits on a condition variable, as typically, in Java programs, a thread waits on a condition variable when it does not have anything to do. This observation is especially true for a server that defines a large pool of threads to handle requests, but where normally only a small portion of the threads are active at any given time. The waiting time is thus not essential to the computation of the application and including it would drastically reduce the CSP, making difficult the identification of phases in which threads do not progress. In contrast, we include in the running times the time where a thread is blocked for other reasons. For example, let us consider an application that spends most of its time in I/O outside any critical section, and that only rarely blocks to acquire a lock. If we do not consider the I/O time, we will report a high CSP, even though the lock is not the bottleneck. Likewise, if we consider the opposite scenario with an application that spends much time blocked in I/O while a lock is held, not counting the I/O time would lead to an underestimated CSP. Finally, we include the scheduled-out time in both the acquiring time and the running time. The probability of being scheduled out while acquiring a lock is the same as the probability of being scheduled out at any other time in the execution, and thus has no impact on the ratio between the acquiring time and the accumulated running time.

As a consequence of our definition, if the CSP becomes large, it means that the threads of the application are not able to execute for long periods of time because they are blocked on the lock. For the generalized ping-pong scenario (Figure 2), in the case where there are only the two ping-pong threads, Free Lunch reports a CSP of 50% because each thread is blocked 50% of the time (large red rectangles). This CSP measurement is satisfactory because it indicates that only half of threads execute at any given time. If we consider more threads, the accumulated running time of the threads will increase, and thus the CSP will decrease. For example, with 48 other threads, Free Lunch will report that the application spends only 2% of its time in lock ac-

quisition, reflecting the fact that the lock does not prevent application progress. For the fork-join scenario (Figure 3), Free Lunch will report a CSP equal to the sum of the times spent while blocked (large red rectangles) divided by the sum of the running times of the threads. As expected, the Free Lunch metric increases when the processing time of the workers decreases, thus indicating that the threads spend more time blocked because of the lock.

3.2 Measurement interval

In order to identify the phases of high CSP of an application, Free Lunch computes the CSP of each lock over a measurement interval. Calibrating the duration of the measurement interval has to take two contradictory constraints into account. On the one hand, the measurement interval has to be small enough to identify the phases of an application. If the measurement interval is large as compared to the duration of a phase in which there is a high CSP, the measured CSP will be negligible and Free Lunch will be unable to identify the high CSP phase. On the other hand, if the measurement interval is too small, the presence of a few blocked threads during the interval can result in a high CSP value, even if there is little pressure on critical sections. In this case, Free Lunch will identify a lot of phases of very high CSP, hiding the actual high CSP phases with a lot of false positive reports.

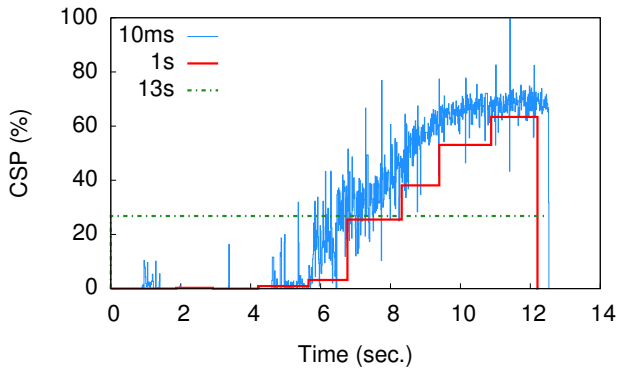


Figure 4. CSP depending on the minimal measurement interval for the Xalan application.

We have tested a range of intervals on the Xalan application from the DaCapo 9.12 benchmark suite. This application is an XSLT parser transforming XML documents into HTML. Xalan exhibits a high CSP phase in the second half of its execution caused by a lot of synchronized accesses to a hash table. Figure 4 reports the evolution of the CSP over time. With a very small measurement interval of 10ms, the CSP varies a lot between successive measurement points. In this case, the lock bounces back and forth from being contended (high points) to being less contended (low points). At the other extreme, when the measurement interval is approximately equal to the execution time (13s), the CSP is averaged over the whole run, hiding the phases. With a measurement interval of 1s, we can observe that (i) the application has a high CSP during the second half of the run with a value that reaches 64%, (ii) the CSP remains relatively stable between two measurement intervals.

Based on the above experiments, we conclude that 1s is a good compromise, as this measurement interval is large enough to stabilize the CSP value. Moreover, if a high CSP phase is shorter than 1s, it is likely that the user will not notice any degradation in responsiveness.

3.3 Free Lunch reports

To further help developers identify the source of high CSP, Free Lunch reports not only the identity of the affected locks, but also, for each lock, an execution path that led to its acquisition. Free Lunch obtains this information by traversing the runtime stack. As traversing the runtime stack is expensive, we have decided to record only a single stack trace, the one that leads to the execution of the acquire operation that causes the monitor to be inflated for the first time. Previous work [2] and our experience in analyzing the Java programs described in Section 6.2 shows that only a single call stack is generally sufficient to understand why a lock impedes thread progress.

3.4 Limitations of our design

A limitation of our design is that Free Lunch only takes into account lock acquisition time as being detrimental to thread progress. Thus, it may report a low CSP in a case where locks are rarely used but many threads are prevented from progressing due to ad hoc synchronization [40] or lock-free algorithms [27]. We leave this issue to future work.

4. Free Lunch implementation

This section presents the implementation details of Free Lunch in the Hotspot 7 JVM for an amd64 architecture. We first describe how Free Lunch measures the different times required to compute the CSP. Then, we present how Free Lunch efficiently collects the relevant information. Finally, we present some limitations of our implementation.

4.1 Time measurement

Free Lunch has to compute the cumulated time spent by all the threads on acquiring each lock and the cumulated running time of all the threads (see Figure 5). Below, we describe how Free Lunch computes these times.

Acquiring time. The acquiring time is the time spent by a thread in acquiring the lock. It is computed on a per lock basis. For this, we have modified the JVM lock acquisition method to record the time before and the time after an acquisition in local variables. A challenge is then where to store this information for further computation. Indeed, we have found that one of the causes of the high runtime overhead of HPROF (see Section 5.1.2) is the use of a map that associates

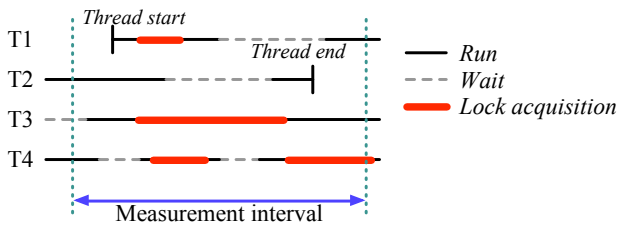


Figure 5. Time periods relevant to the CSP computation.

each Java object to its profiling data. As was for example proposed in RaceTrack [42], Free Lunch avoids this cost by storing the profiling data directly in the structure that represents the profiled entity. Technically, Free Lunch records the acquiring time in a field added to the monitor structure of the JVM. A thread updates this value with its locally calculated acquiring time only when it has already acquired the lock, making it unnecessary to introduce another lock to protect this information.

To accurately obtain a representation of the current time, Free Lunch uses the x86 instruction `rdtsc`, which retrieves the number of elapsed cycles since the last processor restart. The `rdtsc` instruction is not completely reliable: it is synchronized among all cores of a single CPU, but not between CPUs. However, we have empirically observed that the drift between CPUs is negligible as compared to the time scales we consider. A second issue with `rdtsc` is that, as most x86 architectures support instruction reordering, there is, in principle, a danger that the order of `rdtsc` and the lock acquisition operation could be interchanged. To address this issue, general-purpose profilers that use `rdtsc`, such as PAPI [10], introduce an additional costly instruction to prevent reordering. Fortunately, a Java lock acquisition triggers a full memory barrier [29], across which the x86 architecture never reorders instructions, and thus no such additional instruction is needed.

In summary, obtaining the current time when requesting a lock requires the execution of four x86 assembly instructions including `rdtsc` and registering the time in a local variable. Obtaining the current time after acquiring the lock, computing the elapsed lock acquisition time, and storing it in the lock structure require the execution of seven x86 assembly instructions.

A potential limitation of our strategy of storing the acquiring time in the monitor structure is that this structure is only present for inflated monitors. Free Lunch thus collects no information when the monitor is deflated. Acquiring a flat lock, however, does not block the thread, and thus not counting the acquiring time in this case does not change the result.

Computation of running time. As presented in Section 3.1, our notion of running time does not include wait time on condition variables, but does include time when threads are scheduled out and blocked. As such, it does not correspond to the time provided by standard system tools. For this reason, we have chosen to measure the running time directly in the Java virtual machine. In practice, there are two ways for a thread to wait on a condition variable: either by calling the `wait()` method on a monitor, or by calling the `park()` method from the `sun.misc.Unsafe` class. To exclude the waiting times, Free Lunch records the current time just before and after a call to one of these functions, and stores their difference in a thread-local variable. At the end of the measurement interval, Free Lunch computes the running time of the thread by subtracting this waiting time from the time where the thread exists in the measurement interval.

4.2 CSP computation

Free Lunch computes the CSP at the end of each measurement interval. For this, Free Lunch has to visit all of the threads to sum up their running times. Additionally, Free Lunch has to visit all of the monitor structures to retrieve the lock acquiring time. For each lock, the CSP is then computed by dividing the sum of the acquiring times by the sum of the running times.

To avoid introducing a new visit to each of the threads and monitors, Free Lunch leverages the visits already performed by the JVM during the optimized lock algorithm presented in Section 2.1. The JVM regularly inspects each of the locks to possibly deflate them, and this inspection requires that all Java application threads be suspended. Since suspending the threads already requires a full traversal of the threads, Free Lunch leverages this traversal to compute the accumulated running times. Free Lunch also leverages the traversal of all the monitors performed during the deflation phase to compute their CSP.

Our design makes the measurement interval approximate because Free Lunch only computes the CSP during the next deflation phase after the end of the measurement interval. Deflation is performed when Hotspot suspends the application to collect memory, deoptimize the code of a method or redefine a class. After the initial bootstrap phase, however, collecting memory is often the only one of these operations that is regularly performed. This may incur a significant delay in the case of an application that rarely allocates memory. To address this issue, we have added an option to Free Lunch that forces Hotspot to regularly suspend the application, according to the measurement interval.[3] For most of our evaluated applications, however, we have observed that a deflation phase is performed roughly every few tens of milliseconds, which is negligible as compared to our measurement interval of one second.

4.3 Limitations of our implementation

Storing profiling data inside the monitor data structure in Hotspot 7 is not completely reliable, because deflation can

[3] We have used this option for the experiment presented in Figure 4.

break the association between a Java object and its monitor structure at any time, causing the data to be lost. Thus, Free Lunch manages a map that associates every Java object memory address to its associated monitor. During deflation, Free Lunch adds the current monitor to that map. When the lock becomes contended again, the inflation mechanism checks this map to see if a monitor was previously associated with the Java object being inflated. This map is only accessed during inflation and deflation, which are rare events, typically far less frequent than lock acquisition.

Our solution to keep the association between a Java object memory address and its associated monitor is, however, not sufficient in the case of a copying collector [22]. Such a collector can move the object to a different address while the monitor is deflated. In this case, Free Lunch will be unable to find the old monitor. A solution could be to update the map when an object is copied during the collection. We have not implemented this solution because we think that it would lead to a huge slowdown of the garbage collector, as every object would have to be checked.

We have, however, observed that having a deflation of the monitor followed by a copy of the object and then a new inflation of the monitor within a single phase is extremely rare in practice. Indeed, a monitor is deflated when it is no longer contended and thus a deflation will mostly happen between high CSP phases. As a consequence, the identification of a high CSP phase is not altered by this phenomenon. In the case of multiple CSP phases for a single lock, the developer can, however, receive multiple high CSP phase reports indicating different locks. We do not think that this is an issue, because the developer will easily see from the code that all of the reports relate to a single lock.

5. Performance evaluation

We now evaluate the performance of Free Lunch as compared to the existing profilers for OpenJDK: the version of HPROF shipped with OpenJDK version 7, Yourkit 12.0.5, JProfiler 8.0 and MSDK 2.5. As Free Lunch is implemented in Hotspot, we do not compare it with the three profilers for the IBM J9 VM because Hotspot and the IBM J9 VM have incomparable performance.

We first compare the overhead of Free Lunch to that of the other profilers, and then study the cost of the individual design choices of Free Lunch. All of our experiments were performed on a 48-core 2.2GHz AMD Magny-Cours machine having 256GB of RAM. The system runs a Linux 3.2.0 64-bit kernel from Ubuntu 12.04.

5.1 Profiler overhead

We compare the overhead of Free Lunch to that of HPROF, Yourkit, JProfiler and MSDK running in lock profiling mode, on the 11 applications from the DaCapo 9.12 benchmark suite [5], the 19 applications from the SPECjvm2008 benchmark suite [37], and the SPECjbb2005 benchmark [36]. For DaCapo, we run each application 20 times with 10 iterations, and take the average execution time of the last iteration on each run. For SPECjvm2008, we set up each application to run a warmup of 120s followed by 20 iterations of 240s each. For SPECjbb2005, we run 20 times an experiment that uses 48 warehouses and runs for 240s with a warmup of 120s. For SPECjvm2008 and SPECjbb2005, we report the average rate of operations completed per minute. Note that some of the benchmarks cannot be run with some of the profilers: H2 does not run with Yourkit and MSDK, Tradebeans does not run with Yourkit, Compiler.compiler does not run with MSDK, and Avrora and Derby do not run with HPROF.

5.1.1 Overall performance results

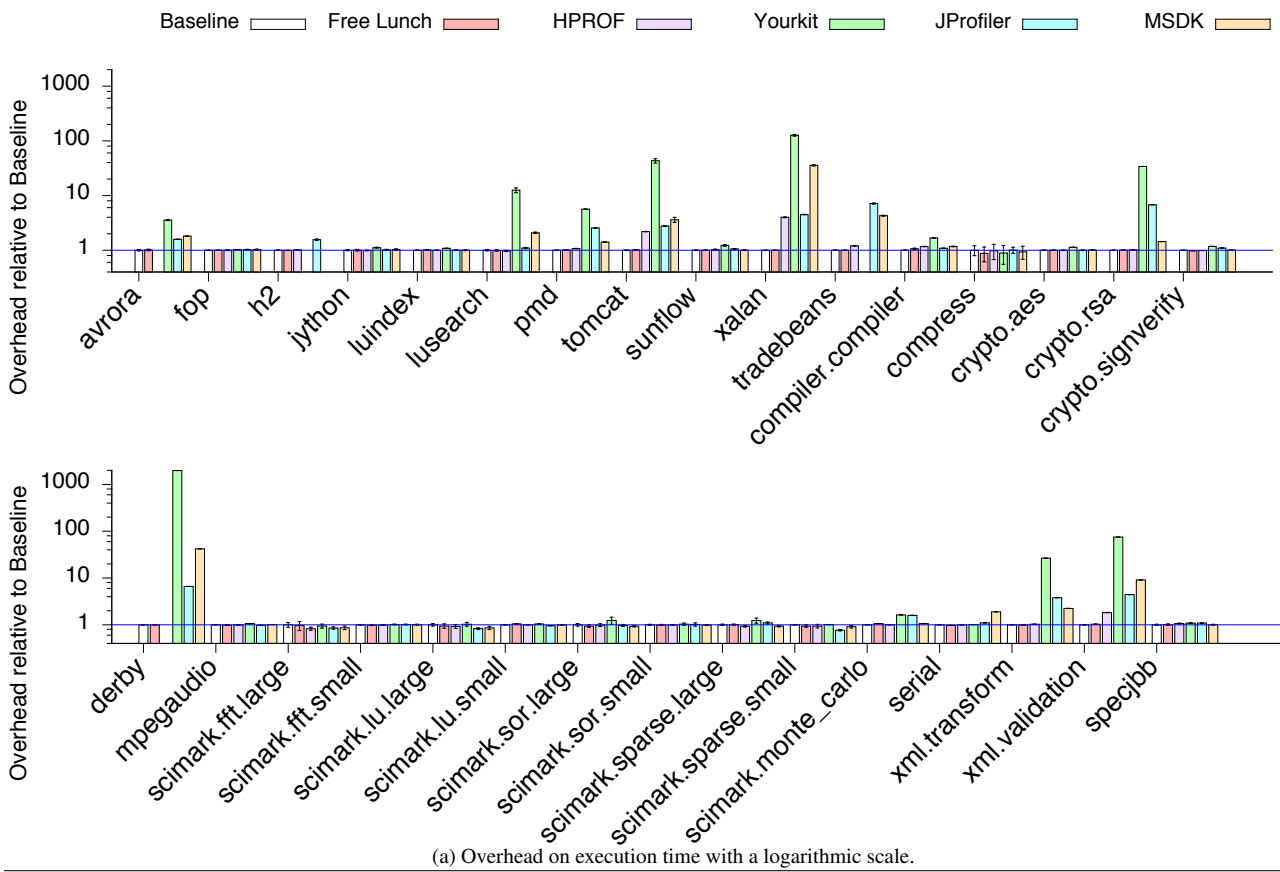
Figure 6 presents the overhead incurred by each of the profilers, as compared to the baseline Hotspot JVM with no profiling, and the standard deviation around this overhead. The results are presented in two ways due to their wide variations. Figure 6.a presents the complete results, on a logarithmic scale, while Figure 6.b focuses on the case between 20% speedup and 60% slowdown.

Figure 6.a shows that the overhead of HPROF can be up to 4 times, that of Yourkit up to 1980 times, that of JProfiler up to 7 times and that of MSDK up to 42 times. Figure 6.b shows that for all applications, the average overhead of Free Lunch is always below 6%. For some of the applications, using a profiler seems to increase the performance. These results are not conclusive because of the large standard deviation.

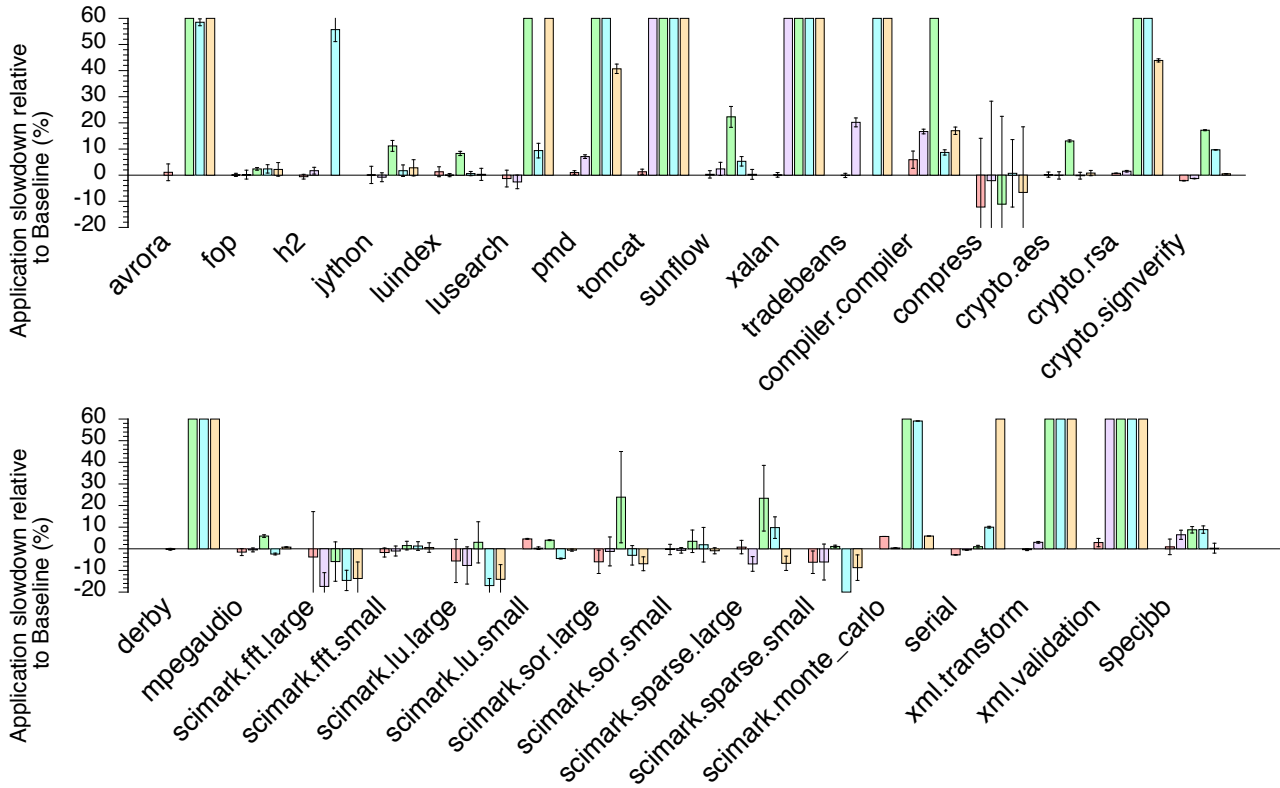
In a multicore setting, as we have here, a common source of large overhead is scalability issues. In order to evaluate the impact of scalability on profiling, we perform additional experiments, using HPROF, which has the least maximum overhead of the existing profilers. We compare HPROF to Hotspot without profiling on the Xalan benchmark in two configurations: 2 threads on 2 cores, and 48 threads on 2 cores. In both cases, the overhead caused by the profiler is around 1%, showing that when the number of cores is small the number of threads has only a marginal impact on profiler performance. Then, we perform the same tests on Xalan with 48 threads on 48 cores. In this case, Xalan runs 4 times slower. These results suggest that, at least in the case of HPROF, the overhead mainly depends on the number of cores.

5.1.2 Detailed analysis of HPROF

We now examine the design of HPROF in more detail, to identify the design decisions that lead to poor scalability. Xalan is the application for which HPROF introduces the most overhead. On this application, we have found that the main issue, amounting to roughly 90% of the overhead, is in the use of locks, in supporting general-purpose profiling and in implementing a map from objects to profiling data.



(a) Overhead on execution time with a logarithmic scale.



(b) Overhead on execution time, limited to between 80% and 160% (zoom of (a)).

Figure 6. Overhead on execution time compared to baseline.

Supporting general-purpose profiling. HPROF, like the other existing profilers, is implemented outside the JVM, relying on JVMTI [24], a standard Java interface that provides data about the state of the JVM. To use JVMTI, a profiler registers two event handlers through the JVMTI API: one that is called before a thread is suspended because it tries to acquire a lock that is already held, and another that is called after the thread has acquired the lock.

When the JVM terminates, HPROF has to dump a coherent view of the collected data. As HPROF is a general-purpose profiler, some event handlers may collect multiple types of information. To ensure that the dumped information is consistent, HPROF requires that no handler be executing while the dump is being prepared. HPROF addresses this issue by continuously keeping track of how many threads are currently executing any JVMTI event handler, and by only dumping the profiling data when this counter is zero. HPROF protects this counter with a single lock that is acquired twice on each fired event, once to increment the counter and once to decrement it.

To measure the cost of the management of this counter, we have performed an experiment using a version of HPROF in which we have removed all of the code in the JVMTI handlers except that relating to the counter and its lock. This experiment shows that the lock acquisition and release operations account for roughly 60% of the overhead of HPROF on Xalan, making this lock a bottleneck at high core count. Note that Free Lunch does not incur this cost because it only supports lock profiling, and a lock operation cannot take place concurrently with the termination of the JVM.

Mapping objects to profiling data. HPROF collects lock profiling information in terms of the top four stack frames leading to a lock acquisition or release event and the class of the locked object. For this, on each lock acquisition or release event, HPROF:

1. Obtains the top four stack frames by invoking a function of the JVM;
2. Obtains the class of the object involved in the lock operation by invoking a function of the JVM;
3. Computes an identifier based on these stack frames and the class;
4. Accesses a global map to retrieve and possibly add the profiling entry associated to the identifier;
5. Accumulates the acquiring time in the profiling entry.

We have evaluated the costs of these different steps, and found that roughly 30% of the overhead of HPROF on Xalan is caused by the access to the map (step 4), and 10% is caused by the other steps. This large overhead during map access is caused by the use of a lock to protect the access to the map, which becomes the second bottleneck at high core count. In contrast, Free Lunch does not incur this overhead because it directly stores the profiling data in the monitor structure of Hotspot, and thus does not require a map and the associated lock to retrieve the profiling entries.

5.2 Free Lunch overhead

We have seen that Free Lunch does not incur the major overheads of HPROF due to their different locking strategies. However, there are other differences in the features of Free Lunch and HPROF that may impact performance. In order to understand the performance impact of these feature differences, we require a baseline that does not include the high locking overhead identified in HPROF in the previous section. Thus, we first create OptHPROF, a lock profiler that collects the same information as HPROF, but that eliminates almost all of HPROF's locking overhead, and then we compare the performance impact of adding the specific features of Free Lunch to OptHPROF, one by one.

5.2.1 OptHPROF

To make our baseline, OptHPROF, for comparison with Free Lunch, we remove the two main bottlenecks presented in Section 5.1.2. First, we simply eliminate the lock that protects the shared counter. As previously noted, this counter is not needed in a lock profiler. Second, for the map that associates an object to its profiling data, we have implemented an optimized version that uses a fine-grain locking scheme, inspired by the lock-free implementation of hash maps found in `java.util.concurrent` [27].

The key observation behind our map implementation is that the profiling data accumulates across the entire execution of the application, and thus no information is ever removed. We represent the map as a hash table, implemented as a non-resizable array of linked lists, where each list holds the set of entries with the same hash code. A read involves retrieving the list associated with the desired profiling entry and searching for the entry in this list. Because the array is not resizable and because no profiling entry is ever removed, a list, whenever obtained, always contains valid entries. Thus, there is no need to acquire a lock when a thread reads the map. A write may involve adding a new entry to the map. The new entry is placed at the beginning of the associated list. Doing so requires taking a lock on the relevant list, to ensure that two colliding profiling entries are not added at the same time. As in Free Lunch, profiling data are recorded in a profiling entry after the lock associated with the profiling entry is acquired, and thus no additional locking is required.

The map itself is mostly accessed for reads: a write is only required the first time a profiling entry is added to the map, which is much less frequent than adding new profiling information to an existing entry. Likewise, it is rare that two threads need to access the same profiling entry at the same time. Thus, the locks found in OptHPROF are not likely to be contended, allowing OptHPROF to scale with the profiled application.

Figure 7 reports the overhead of OptHPROF on Avrora, H2, PMD, Sunflow, Tomcat, Tradebeans, Xalan and Xml.Va-

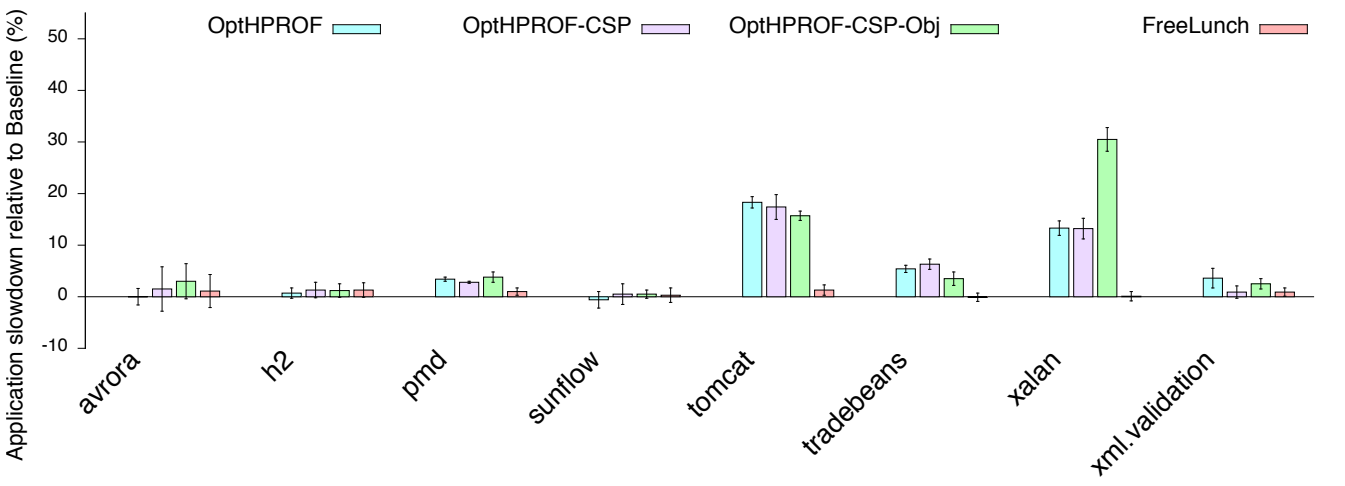


Figure 7. Overhead on execution time compared to baseline.

lidation, which are the applications that are most slowed down by HPROF. By eliminating the counter lock and by using a more scalable map data structure, the worst-case overhead of OptHPROF is 18.3% with Tomcat, which approaches the worst-case overhead of Free Lunch, of 6%.

5.2.2 Free Lunch features

The main features of Free Lunch that are not found in OptHPROF, and thus that are not found in HPROF, are as follows:

- **Metric**: Free Lunch supports profiling of phases, and thus computes its metric at regular intervals, while OptHPROF computes its metric only at the end of the run. Furthermore, OptHPROF only reports the acquisition time of a lock divided by the total acquisition time of any lock, while Free Lunch reports the CSP, *i.e.,* the acquisition time of a lock divided by the accumulated running time of the threads of the application.
- **Profiling granularity**: Free Lunch indexes profiling information at the object level, while OptHPROF indexes profiling information by the object's class and the top four stack frames at the time of the lock operation. OptHPROF's strategy makes it possible to identify the critical section in which a problem is observed, and the context in which that critical section was reached, but it risks conflating information from multiple objects of the same class, and hiding locking issues that are dispersed across multiple critical sections. In contrast, Free Lunch only collects a stack trace at the first contended acquisition of a given object's lock, which may not be the critical section in which contention occurs, but unifies all of the profiling information about a given object within the current time interval.
- **Integration with the JVM**: Free Lunch directly reuses the internal representation of a monitor inside the JVM to store the profiling data, while OptHPROF is independent of the JVM and has to access an external map for each lock operation.

Experiment	Metric	Stack trace	Out-VM	Data structure
HPROF	HPROF	Each acquisition	Yes	Not optimized
OptHPROF	HPROF	Each acquisition	Yes	Optimized
OptHPROF-CSP	CSP	Each acquisition	Yes	Optimized
OptHPROF-CSP-Obj	CSP	First acquisition	Yes	Optimized
Free Lunch	CSP	First acquisition	No	Optimized

Table 2. Experiments conducted to understand Free Lunch.

We evaluate each of these differences in terms of the set of experiments described in Table 2. Each experiment involves creating a variant of OptHPROF that mimics Free Lunch in one or more of the above aspects, Figure 7 reports the overhead introduced by each of the variants, along with the standard deviation on 5 runs, with the same applications Avrora, H2, PMD, Sunflow, Tomcat, Tradebeans, Xalan and Xml.Validation. We now analyze the implementations of the above variants and their results in detail.

OptHPROF-CSP: using phases and the CSP instead of the HPROF's metric. To implement OptHPROF-CSP, we modify the implementation of OptHPROF to periodically compute the CSP rather than computing HPROF's metric once at the end of the run. Several issues must be addressed. First, the CSP is computed in terms of the lock acquisition time and the running time. Of these, only the lock acquisition time is already computed by OptHPROF. To compute the running time, we extend OptHPROF to intercept the calls to the wait functions and to the thread creation and destruction functions through JVMTI events. Finally, OptHPROF-CSP cannot piggy-back on the garbage collector, as done by Free Lunch, to compute the CSP periodically, because GC events are not made available via JVMTI. Instead, OptHPROF-CSP defines a thread, woken up every second, to perform the computation.

As presented in Figure 7, regularly computing the CSP instead of computing the HPROF metric at the end of the run does not introduce a significant overhead. In the worst case, regularly computing the CSP increases the overhead by 1.5% and, in the best case, it reduces the overhead by 2.7%. Thus, neither the choice of which of these metrics to compute nor the frequency of the computation has an impact on performance.

OptHPROF-CSP-Obj: profiling granularity. To implement OptHPROF-CSP-Obj, we modify the implementation of OptHPROF-CSP to index the profiling entries by object rather than by class and stack frames. For this, we use the internal hashcode embedded in any Java object as the profiling entry identifier. To further simulate the behavior of Free Lunch, we also extend OptHPROF-CSP to record a full stack trace at the first acquisition of each lock.

As presented in Figure 7, we can see that, except for Xalan, recording a full stack trace at the first lock acquisition or systematically recording the first four frames at each lock acquisition does not have a significant impact on the performance. In the best case, OptHPROF-CSP-Obj increases the performance by 2.8% and in the worst case, except for Xalan, it reduces the performance by 1.6%.

For Xalan, however, not recording the stack frames at each lock acquisition adds a significant overhead of 17%. This result is unexpected because computing a hashcode only consists of reading the object header, which should take less time than recording four stack frames. Indeed, we have measured that, on average, OptHPROF-CSP adds an overhead of roughly 50,000 cycles before each lock acquisition on Xalan, while OptHPROF-CSP-Obj only adds an overhead of roughly 2,500 cycles.

To better understand this result, we have conducted another experiment, in which we explore the impact of changing the delay before the lock acquisition on the performance of Xalan. Starting from the implementation of OptHPROF-CSP, we replace the JVMTI handler code before the lock acquisition by a delay of varying length, leaving the JVMTI handler code of OptHPROF-CSP after the lock acquisition unchanged. Figure 8 reports the overhead caused by the varying delay as compared to an execution of Xalan without any instrumentation (baseline). We first observe that the instrumentation of OptHPROF-CSP after the lock acquisition slows down the application by roughly 30%. Subsequently, the impact of the delay varies greatly in the zones marked A, B, and C in the graph. In zone A, from a delay of 1 cycle to a delay of 50,000 cycles, the overhead slightly decreases as the delay increases. This counterintuitive result is due to the fact that spinlocks and POSIX locks, which are used by Java to implement synchronization, saturate the memory buses when many threads try to acquire a lock simultaneously [28]. Increasing the delay gradually reduces the contention on the memory buses and the resulting performance improvement outweighs our introduced delay. In zone B, from a delay of 50,000 cycles to a delay of 10^6 cycles, the problem of memory bus saturation is reduced significantly, leading to a huge reduction in the overall overhead induced by the delay and indeed an improvement over the performance of Xalan without profiling, which itself suffers from saturation of the memory buses. Finally, in zone C, the buses are no longer saturated and the overhead increases linearly with the delay, as expected.

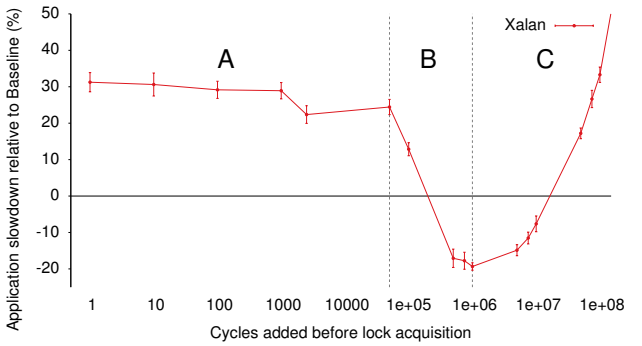


Figure 8. Overhead of Xalan with a varying delay before lock acquisition.

In our context, by not recording the first four stack frames, we reduce the delay between two lock acquisitions, which further saturates the buses, and thus leads to worse performance. It should be noted that in the case of OptHPROF-CSP, the code executed before each lock acquisition may involve cache misses, while the wait introduced in the above experiment does not. The cycle count thresholds separating zones A, B, and C are thus not exactly comparable.

Free Lunch: integration with the JVM. OptHPROF-CSP-Obj is a profiler that has essentially the same functionality as Free Lunch, but is implemented outside of the JVM. By comparing it with Free Lunch, we can thus identify the benefit of leveraging the internal monitor structure of the JVM to store the profiling data.

As presented in Figure 7, leveraging the internal data structures of the JVM significantly decreases the overhead caused by the use of a profiler, especially on Tomcat, Tradebeans and Xalan, the three applications that are the most slowed down by OptHPROF-CSP-Obj. For Tomcat, the overhead decreases from 15.7% with OptHPROF-CSP-Obj to 1.3% with Free Lunch, for Tradebeans from 3.5% to less than 0.1%, and for Xalan from 30.5% to less than 0.1%.

6. Using Free Lunch to analyze applications

We now experimentally validate our analysis of the metrics presented in Section 2.2 and report our results when using Free Lunch to analyze the lock behavior of the applications considered in Section 5 as well as Cassandra 1.0.0 [26].

Contention metric	2 threads	48 threads	Profiler
CSP	49.9%	2.1%	Free Lunch
Acquiring time of a lock / Acquiring time of all locks	99%	99%	HPROF
Acquiring time of a lock / Elapsed time	96.7%	96.7%	JProfiler
Total CS time of a lock / # of acquisitions	2.4ms	2.4ms	MSDK

Table 3. Evaluation of contention metrics on the ping-pong micro-benchmark.

All evaluations are performed on the machine described in Section 5.

6.1 Micro-benchmarks

We instantiate the scenarios described in Section 2.2 into micro-benchmarks and use them to compare the ability of the CSP metric and the other metrics to indicate the impact of locks on thread progress.

We first consider the ping-pong micro-benchmark, instantiating the micro-benchmark such that each ping-pong thread spends 1ms in the critical section on each iteration. We execute the micro-benchmark for 30s, with 2 and 48 threads. The results are presented in Table 3.

For this micro-benchmark, we first study the profilers that rely on the acquiring time. On the ping-pong scenario, for both 2 and 48 threads, HPROF reports that 99% of the acquiring time of any lock is spent to acquire the ping-pong lock and 1% is spent to acquire internal locks of the Java library during the bootstrap of the application. Thus, as anticipated by our theoretical study, the result reported by HPROF does not change with the number of threads. JProfiler reports the time spent in acquiring each lock and the elapsed time of the application: the acquiring time equals 96.7% of the elapsed time with 2 or 48 threads. This result also confirms our theoretical analysis. Thus, neither of these metrics decreases when the number of threads increases. On the other hand, Free Lunch reports a CSP of 49.9% with 2 threads and a CSP of 2.1% with 48 threads. Thus, it correctly indicates when the lock impedes the progress of threads.

We next study the profilers that rely on the critical section time. MSDK's metric divides this time by the total number of acquisitions. On the ping-pong micro-benchmark, it reports a value of 2.4ms with both 2 and 48 threads (see Table 3). Thus, again, as predicted by our theoretical analysis, the result does not decrease when the number of threads increases.

We then turn to the fork-join micro-benchmark. We also execute this micro-benchmark for 30s, with 1 master thread and 47 worker threads. We vary the processing time of the workers from 50ms to 700ms. The results are presented in Figure 9.

For this micro-benchmark, we compare Free Lunch with Health Center, which relies on the number of failed acquisitions. As shown in Figure 9, the CSP reported by Free Lunch

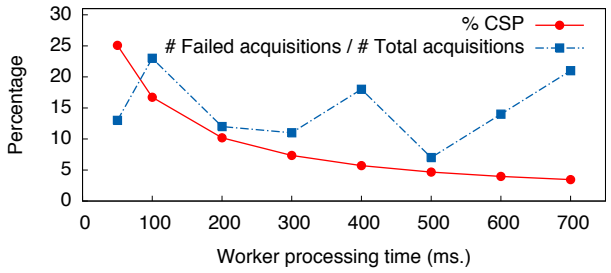


Figure 9. Comparison of the Free Lunch CSP metric and the Health Center metric on the fork-join micro-benchmark.

decreases with the processing time of the workers. On the other hand, the number of failures divided by the number of acquisitions reported by Health Center oscillates between 7 and 23%, depending on the processing time, and does not decrease when the processing time increases. This result corresponds to the theoretical study presented in Section 2.2: the number of failures divided by the number of acquisitions is not related to the processing time of the workers, and thus the progress of the threads. Notice that according to our theoretical study, Health Center should report a constant value of $5/9$ (56%). That value does not account for the fact that the Linux scheduler has to elect the workers when they are woken up by the master. This election time avoids lock acquisition failures when a thread is elected after the already awakened threads have released their lock. On the other hand, as a condition variable may not wake up the waiting threads in FIFO order, some failed acquisitions can occur during the join phase.

6.2 Analysis of lock CSP

This section presents a detailed analysis of the CSP of the locks used by the applications from the DaCapo 9.12 benchmark suite [5], the SPECjvm2008 [37] benchmark suite, and the SPECjbb2005 [36] benchmark. We first consider the case where the measurement interval is equal to the running time of the application, giving the average CSP over the whole run. Table 4 lists the locks with an average CSP greater than 5% in this case. Figure 10 then presents the evolution of the CSP of the same locks with a measurement interval of 1s. Note that the average CSP over the whole run (Table 4) is not equal to the average of the CSPs of each individual measurement interval (Figure 10), because of changes in the number of threads in each measurement interval. For example, a high CSP with only two running threads during a measurement interval becomes negligible when averaged over two measurement intervals if many threads are running in the second interval. The remainder of this section analyzes in detail these CSP values.

H2 is an in-memory database. The lock associated with an `org.h2.Database` object has an average CSP of 62.3%. H2 uses this lock to ensure that client requests are pro-

Benchmark	Java class of the object with highest CSP	CSP
H2	org.h2.engine.Database	62.3%
Avrora	java.lang.Class	48.4%
PMD	org.dacapo.harness.DacapoClassLoader	25.4%
Xalan	java.util.Hashtable	20.4%
Sunflow	org.sunflow.core.Geometry	6.2%
Tradebeans	org.h2.engine.Database	6.0%

Table 4. CSP averaged during the whole run.

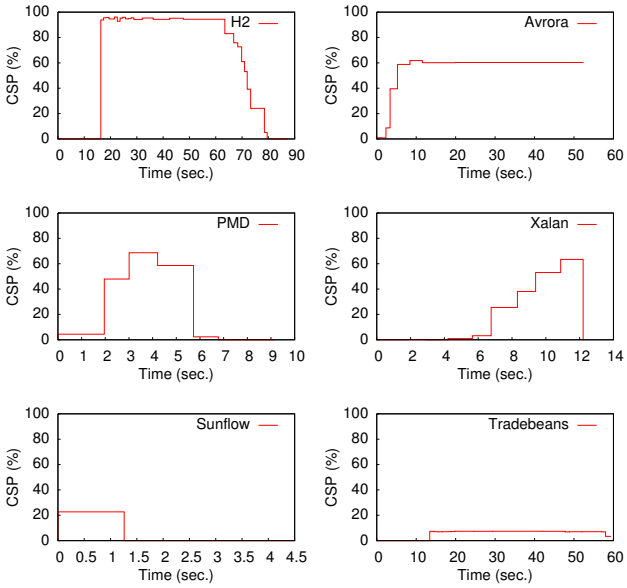


Figure 10. CSP with a measurement interval of 1s.

cessed sequentially; thus, the more clients send requests to the database, the more clients try to acquire the lock. As shown in Figure 10, H2 exhibits 3 distinct phases. The first phase (from 0s to 16s) presents no CSP at all: in this phase the main thread of the application populates the database, thus no CSP occurs for accessing the database. The second phase (from 16s to 79s) shows a CSP between 92% and 96%: clients are sending requests to the database, thus inducing contention on the database lock. The CSP decreases at the end of the phase, going from 92% to 0%, when clients have finished their requests to the database. The purpose of the last phase (from 79s to the end) is to revert the database back to its original state, which is again done only by the main thread and thus induces no CSP. This application is inherently not scalable because requests are processed sequentially. Deep modifications would be required to improve performance.

Avrora is a simulation and analysis framework. The lock associated with a `java.lang.Class` object has an average CSP of 48.4%. Avrora uses this lock to serialize its output to the terminal. As shown in Figure 10, Avrora exhibits a high CSP phase, from 2.3s to the end of the application, where application threads write results to a file. There seems to be no simple solution to remove this lock because interleaving outputs from different threads would lead to an inconsistent result.

PMD is a source code analyzer. The lock associated with an `org.dacapo.harness.DacapoClassLoader` object has an average CSP of 25.4%. This object is used to load new classes during execution. As shown in Figure 10, a high CSP phase begins at 2s and terminates at 5.7s, while the application terminates at 9.2s. During the high CSP phase, PMD stresses the class loader because all the threads are trying to load the same classes. Removing this bottleneck is likely to be hard because the classes have to be loaded serially.

Xalan is a XSLT parser transforming XML documents into HTML. The lock associated with a `java.util.Hashtable` object has an average CSP of 20.4%. `java.util.Hashtable` uses this lock to ensure mutual exclusion on each access to the hashtable, leading to a bottleneck. As shown in Figure 10, during a first phase (from 0s to 6.8s) only one thread fills the hashtable, and therefore the CSP is negligible. However, during the second phase (from 6.8s to the end of application), all the threads of the application are accessing the hashtable, increasing the CSP up to 64.3%. This high CSP phase is underestimated when the CSP is averaged over the whole run, making it difficult to identify without separating phases. We reimplemented the hash table using `java.util.concurrent.ConcurrentHashMap`, which does not rely on locks. This change required modifying a single line of code, and improved the baseline application execution time by 15%. This analysis shows that the information generated by Free Lunch can help developers in practice.

Sunflow is an image rendering application. The lock associated with an `org.sunflow.core.Geometry` object has an average CSP of 5.8%. As shown in Figure 10, Sunflow exhibits a moderate CSP peak at the beginning of its execution. This occurs during the tesselation of 3D objects, which must be done in mutual exclusion. Since the number of 3D objects is small as compared to the number of threads, many threads block, waiting for the tesselation to complete. Improving the performance would require parallelizing tessalation computation.

Tradebeans simulates an online stock trading system, and includes H2 to store persistent data. The lock associated with an `org.h2.Database` object has an average CSP of 6.0%. This lock is also the bottleneck reported in the H2 application. As shown in Figure 10, a phase with a small CSP starts at 13.4s and persists until the application terminates. As already explained, deep modifications would be required in H2 to improve performance.

6.3 Cassandra

Cassandra [26] is a distributed on-disk NoSQL database, with an architecture based on Google's BigTable [7] and

Amazon's Dynamo [8] databases. It provides no single point of failure, and is meant to be scalable and highly available. Data are partitioned and replicated over the nodes. Durability in Cassandra is ensured by the use of a commit log where it records all modifications. As exploring the whole commit log to answer a request is expensive, Cassandra also has a cache of the state of the database. This cache is partially stored to disk and partially stored in memory. After a crash, a node has to rebuild this cache before answering client requests. For this purpose, it rebuilds the cache that was stored in memory by replaying the modifications from the commit log.

A Cassandra developer reported a lock performance issue in Cassandra 1.0.0.[4] During this phase, the latency was multiplied by twenty. The issue was observed on a configuration where the database is deployed on three nodes with a replication factor of three, and consistency is ensured by a quorum agreement of two replicas. No further information about the configuration is provided. As a result, we were unable to reproduce this problem.

Although we were not able to reproduce the previously reported problem, we were able to use Free Lunch to detect a phase with a high CSP in Cassandra 1.0.0. Using the configuration described above, we created a 10Gb database and then used the YCSB [34] benchmark to stress Cassandra with an update-heavy workload including 50% reads and 50% updates. After 5.5 minutes, we simulated a crash by halting a node and immediately restarting it. During the recovery, Free Lunch reports a high CSP phase of around 50%, with a peak at 52%. The high CSP phase takes place during the commit log replay, which takes 11.4s. Coincidentally, the critical section involved is the same one that caused the previously reported problem in Cassandra 1.0.0. Outside this phase, the CSP for the lock is near 0%. The duration of the high CSP phase is proportional to the size of the log replay, which itself is proportional to the number of modifications before the crash. This result shows that Free Lunch is able to accurately identify variations in CSP during phases in large Java servers. This phase is hidden by other profilers because a Cassandra server has a long running time of many days.

This experiment also illustrates the difficulty of producing and reproducing CSP issues. Indeed, the particular tested scenario is complex to deploy and involves a server crash, which is relatively unusual. For this reason, we think that the probability of encountering the issue during *in-vitro* testing is small, and thus *in-vivo* profiling is essential.

7. Related work

Lock profilers. We have already presented the profilers Health Center [16], HPROF [17], Yourkit [41], MSDK [32], JLM [30], JLA [20], and JProfiler [23] in Sections 2 and 5. Our analysis shows that all of these profilers use metrics that do not provide useful progress information for some synchronization patterns and only report their results at the end of the application, thus hiding phases. For example, none of these profiler metrics highlights the bottlenecks we have observed in Xalan and Cassandra, and, as the high CSPs are hidden in the phases, they cannot identify them. All but Health Center furthermore incur a high performance overhead, and are thus not suitable for *in-vivo* profiling. Health Center is based on sampling, and thus has very low overhead. Nevertheless, Health Center is only available for the IBM J9 JVM. We have observed that J9 without profiling is at least 2 times slower than Hotspot 7 on 9 of our 31 benchmarks. On the Xalan benchmark, which exhibits a high CSP phase, J9 is 7.3 times slower than Hotspot. These differences makes it difficult to compare a profiler that runs on J9 with a profiler that runs on Hotspot.

Inoue et al. [19] have proposed a profiler for flat locks that has been implemented in Health Center. This profiler uses the same metric as Health Center and thus suffers from the same limitations. For C applications, Mutrace [33] and the profiler used in RCL [28] also have the same limitations of the above Java profilers.

WAIT [2] is a tool that uses sampling to diagnose various performance issues in running server-class applications in order to understand the cause of idleness of threads. To measure lock usage, WAIT counts the number of threads blocked while acquiring a lock. The performance impact is proportional to the rate of sampling, which ranges from unnoticeable (1 sample every 1000 seconds) to 59% (1 sample per second). WAIT incurs more overhead than Free Lunch once the sampling rate reaches 1 sample every 20 seconds (8%). As several samples are needed to make sure that the lock contention is sustained, it is likely to miss short lock usage phases like the ones with high CSP in Cassandra or Xalan.

Xian et al. [39] propose to dynamically detect lock contention induced by the OS on Java applications at runtime. Their approach segregates threads that contend for the same lock on the same core and ensures that a lock owner is allowed to run as long as it owns the lock. Therefore, it avoids lock contention induced by OS activities such as thread preemption. This approach is complementary to ours because it focuses on lock contention induced by the OS, whereas Free Lunch focuses on lock contention induced by applications.

Lockmeter [6] is a tool that targets spinlock profiling for the Linux kernel. Like, e.g., Java Lock Monitor [30], Lockmeter reports the time spent in the critical section protected by a spinlock divided by the elapsed time. As shown in Section 2, this metric does not report a useful value on some synchronization patterns.

HPCToolkit [38] is a profiler designed for high performance computing. The authors define a new metric to attribute lock contention to the threads that are responsible for it. This approach is complementary to Free Lunch in the

[4] See `https://issues.apache.org/jira/browse/CASSANDRA-3385` and `https://issues.apache.org/jira/browse/CASSANDRA-3386`.

sense that HPCToolkit attributes lock contention to threads whereas Free Lunch measures lock-related CSP.

Finally, HaLock [18] is a hardware-assisted lock profiler. It relies on a specific hardware component that tracks memory accesses in order to detect heavily used locks. This technique achieves low overhead but requires dedicated hardware.

Other profilers for parallel applications. Capacity planning [31] is a technique used to identify where applications have to be optimized. For that purpose, it breaks down an application into tasks and is able to tell if and how optimizing them can lead to a performance improvement. The developers of capacity planning observe that a critical section can act as bottleneck for many reasons, not all of which are related to the synchronization pattern. For example, if too many threads are running, the owner of a lock can often be scheduled out by the operating system, making the lock appear as a bottleneck. Capacity planning needs inference rules provided by application experts to be able to cut the application into tasks and to correlate the observations to the application source code. On the contrary, Free Lunch focuses on legacy code and does not require any help from the programmer to identify the locks that impede thread progress. Free Lunch thus has a larger applicability, but it only provides raw data, which could be used as a building block for capacity planning.

Bottle Graphs [11] is a profiling tool that is able to graphically illustrate the parallelism of an application. The degree of parallelism is mainly defined as the time where threads are not suspended divided by their execution time. Bottle Graphs reports a macroscopic view of the parallelism of an application, which makes it useful in understanding whether the parallelism of the application could be enhanced and in identifying how each thread contributes to the processing. Free Lunch is complementary to Bottle Graphs, as it is able to indicate whether a lack of parallelism comes from lock usage.

Kalibera et al. [25] define new concurrency metrics, analyze communication patterns of shared Java objects, and apply the new concurrency metrics to the DaCapo benchmarks [5]. They evaluate locking behavior by counting the number of monitor acquisitions and the global locking rate of the application, along with the pattern by which these objects are accessed by threads. This work is complementary to ours, in that it gives a global view of shared-object behavior whereas Free Lunch provides detailed information about CSP for each lock.

Limit [9] provides a lightweight interface to on-chip performance counters. Indeed, the elapsed time obtained using `rdtsc` can be inaccurate when a thread is scheduled out or migrated on a multicore machine. Limit solves this issue by using a dedicated kernel module. In Free Lunch, we do not want to exclude the scheduled out time, and thus we do not need the former feature of Limit. In case of migration, as stated in Section 4.1, we have observed that the drift between the CPUs is not significant.

`Java.util.concurrent` is a Java API that provides lock-free data structures. JUCProfiler (which is part of MSDK [32]) and JProfiler [23] are able to profile such libraries. Free Lunch does not currently provide this type of profiling. We plan to support lock-free data structures in future work.

8. Conclusion

This paper has presented Free Lunch, a new lock profiler especially designed to identify *phases* of high Critical Section Pressure (CSP) *in-vivo*. Using Free Lunch, we have identified phases of high CSP in six applications from the DaCapo benchmark suite, the SpecJVM 2008 benchmark suite and the SpecJBB 2005 benchmark, and a phase of high CSP in Cassandra. Some of these phases are hidden when using existing profilers, which shows that Free Lunch can identify new bottlenecks and reports them back to the developer. Thanks to these reports, we were able to improve the performance of the Xalan application by 15% by modifying a single line of code.

We have evaluated Free Lunch on more than thirty applications and shown that it never degrades the performance by more than 6%. This result shows that Free Lunch could be used *in-vivo* to detect phases where a lock impede the threads' progress with scenarios that would otherwise not necessarily be tested by a developer *in-vitro*.

Acknowledgements

This work was supported in part by the ANR project Infra-JVM.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine project: Building an open source research community. *IBM System Journal*, 2005.

[2] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *OOPSLA*, pages 739–753, 2010.

[3] Apache Tomcat web page. `http://tomcat.apache.org/`, 2014.

[4] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *PLDI*, pages 258–268, 1998.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[6] R. Bryant and J. Hawkes. Lockmeter: Highly-informative instrumentation for spin locks in the Linux kernel. In *4th Annual Linux Showcase & Conference*, pages 271–282, 2000.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[9] J. Demme and S. Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *ISCA*, 2011.

[10] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters. In *IPDPS*. IEEE, 2003.

[11] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle graphs: visualizing scalability bottlenecks in multi-threaded applications. In *OOPSLA*, pages 355–372, 2013.

[12] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. VMKit: A substrate for managed runtime environments. In *VEE*, pages 51–62, 2010.

[13] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *SOSP*, pages 103–116, 2009.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ language specification*. Addison-Wesley, 3rd edition, 2005.

[15] H2 web page. `http://www.h2database.com/`, 2014.

[16] Healthcenter. IBM Health Center. `http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/`, 2014.

[17] HPROF: A heap/CPU profiling tool. `http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html`, 2014.

[18] Y. Huang, Z. Cui, L. Chen, W. Zhang, Y. Bao, and M. Chen. HaLock: hardware-assisted lock contention detection in multithreaded applications. In *PACT*, pages 253–262, 2012.

[19] H. Inoue and T. Nakatani. How a Java VM can get more from a hardware performance monitor. In *OOPSLA*, pages 137–154, 2009.

[20] Java Lock Analyzer. JLA homepage. `http://publib.boulder.ibm.com/infocenter/javasdk/tools/index.jsp?topic=%2Fcom.ibm.java.doc.igaa%2F_1vg0001143f2181-11a9b04924e-7ff9_1001.html`, 2014.

[21] JBoss web page. `https://www.jboss.org/overview/`, 2014.

[22] R. Jones, A. Hosking, and E. Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 1st edition, 2011.

[23] JProfiler home page. `http://www.ej-technologies.com/products/jprofiler/overview.html`, 2014.

[24] JVMTI. Java™ Virtual Machine Tool Interface. `http://docs.oracle.com/javase/6/docs/technotes/guides/jvmti/`, 2014.

[25] T. Kalibera, M. Mole, R. Jones, and J. Vitek. A black-box approach to understanding concurrency in DaCapo. In *OOPSLA*, pages 335–354, 2012.

[26] A. Lakshman and P. Malik. Cassandra: Structured storage system on a P2P network. In *PODC*, 2009.

[27] D. Lea. The java.util.concurrent synchronizer framework. *Sci. Comput. Program.*, pages 293–309, 2005.

[28] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote Core Locking: migrating critical-section execution to improve the performance of multithreaded applications. In *ATC*, pages 65–76. USENIX, 2012.

[29] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, pages 378–391, 2005.

[30] M. Milenkovic, S. Jones, F. Levine, and E. Pineda. Performance inspector tools with instruction tracing and per-thread / function profiling. In *Linux Symposium*, 2008.

[31] N. Mitchell and P. F. Sweeney. On-the-fly capacity planning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA, pages 849–866, 2013.

[32] Multicore SDK. `https://www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUuid=9a29d9f0-11b1-4d29-9359-a6fd9678a2e8`, 2014.

[33] Mutrace. Measuring Lock Contention. `http://0pointer.de/blog/projects/mutrace.html`, 2014.

[34] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. YCSB++: Benchmarking and performance debugging advanced features in scalable table stores. In *SoCC*. ACM, 2011.

[35] Safepoints in Hotspot. `http://blog.ragozin.info/2012/10/safepoints-in-hotspot`, 2014.

[36] SPECjbb2005. `http://www.spec.org/jbb2005/`, 2014.

[37] SPECjvm2008. `http://www.spec.org/jvm2008/`, 2014.

[38] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *PPoPP*, pages 269–280, 2010.

[39] F. Xian, W. Srisa-an, and H. Jiang. Contention-aware scheduler: unlocking execution parallelism in multithreaded Java programs. In *OOPSLA*, pages 163–180, 2008.

[40] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *OSDI*, pages 1–8. USENIX, 2010.

[41] Yourkit. Yourkit home page. `http://www.yourkit.com/`, 2014.

[42] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, pages 221–234, 2005.