Continuous and Efficient Lock Profiling for Java on Multicore Architectures

Florian David

HAL Id: tel-01263203
https://hal.inria.fr/tel-01263203v2

Submitted on 31 Mar 2016

PhD Thesis, Université Pierre et Marie Curie
Whisper team, LIP6/INRIA

Subject: Computer Science
Option: Distributed Systems

Presented by: Florian David

Continuous and Efficient Lock Profiling for Java on Multicore Architectures

*Profilage Continu et Efficient de Verrous
pour Java pour les Architectures Multicœurs*

Presented on 08/07/15 in front of the following jury:

M. Pascal Felber	Université de Neuchâtel (IIUN)	Neuchâtel, Suisse	*Reviewer*
M. Laurent Réveillère	Université de Bordeaux 1 (LaBRI)	Bordeaux, France	*Reviewer*
M. Emmanuel Chailloux	Université Pierre et Marie Curie (LIP6)	Paris, France	*Examiner*
M. Didier Donsez	Université de Grenoble 1 (ERODS)	Grenoble, France	*Examiner*
M. Jens Gustedt	Université de Strasbourg (ICube/INRIA)	Illkirch, France	*Examiner*
M. Gilles Muller	Université Pierre et Marie Curie (LIP6/INRIA)	Paris, France	*Advisor*
M. Gaël Thomas	Telecom SudParis (HP2)	Evry, France	*Advisor*

Remerciements

L'achèvement de cette thèse n'aurait pas été possible sans le soutien, les encouragements ou tout simplement la présence de toutes les personnes qui ont été à mes côtés pendant ces 4 années.

Cette aventure n'aurait jamais pu exister sans Gilles Muller, Gaël Thomas et Julia Lawall. Merci à vous 3 pour m'avoir encadré pendant ces 4 années de thèse et guidé grâce à nos discussions et votre disponibilité sans faille (de jour, comme de nuit :)). J'ai énormément appris à votre contact et même si cela n'a pas été facile tous les jours, je ne regrette en rien cette expérience. En particulier, merci à Gaël pour sa bonne humeur communicative et ses interventions théâtrales dans notre bureau qui avaient le don de me remonter le moral (ainsi que celui des autres thésards) en tout circonstance !

Merci à Emmanuel Chailloux, Didier Donsez et Jens Gustedt pour leur présence à ma soutenance de thèse et en particulier à Laurent Réveillère et Pascal Felber pour avoir accepté d'en être les rapporteurs.

J'ai également eu le privilège d'avoir été accompagné par des personnes qui sont devenues bien plus que de simples collègues de bureau : Sergey Legtchenko (un verre avec toi et Élo et tout de suite, le courant est passé. Merci pour ces week-ends en Bourgogne et à Cambridge, nos discussions sur notre vision de la recherche et pour m'avoir montrer qu'on pouvait faire progresser la science en jouant à Quake 3 ;-)), Thomas Preud'homme (pour ta bonne humeur, pour m'avoir accueilli avec Yan chez vous à Shangaï et pour toutes nos conversations à propos de Linux, des compilateurs ou encore du défi mathématique du vendredi !), Lokesh Gidra (I have really appreciated to discuss, to exchange some tips, and (mainly) to complain about our troubles with the JVM with you :D), Maxime Lorrillere (par ta manière posée et réfléchie d'aborder et de m'aider à résoudre les problèmes ainsi que pour ces vacances mémorables à l'UCPA Chamonix ;-)), Brice Berna (pour tous ces subreddits improbables que l'on a découvert), Rudyar Cortes (toujours disponible pour "un cafecito ?"), Marek Zawirski (thanks for all the (many) opportunities you provide me for slacking off by playing Mario Kart or drinking beers ;-)) et Anissa Lamani (j'ai eu un immense plaisir à venir te voir au Japon et même si l'on se voit rarement, j'apprécie particulièrement cette facilité que l'on a à discuter comme si l'on s'était quitté la veille). Sur le chemin vers la pause café, elle est toujours postée en embuscade, l'œil droit sur son écran et le gauche sur le couloir, j'ai nommé: Laure Millet ! Un grand merci également à Julien Sopena (avec qui j'ai pu démontrer que le transport de données par camion

de cartes micro-SD pouvait être plus rapide que la fibre optique), Olivier Marin (qui a su garder son calme malgré toutes les fois où je lui ai emprunté ses ciseaux) et Jean-Luc Mounier (pour m'avoir aidé avec le Mac/imprimante/wifi/réseau/*insert anything here* et avoir toujours gardé son calme quand la solution était indiqué sur l'intranet).

Sophie, Théo, Rudyar, Julien, Laetitia, Laura, Alex et Marion. On se connait depuis peu mais je me suis tout de suite senti à l'aise avec vous et je sais déjà que l'on sera amené à se revoir pendant longtemps . . .

Tout le groupe BFC de l'INSEAD et en particulier mon groupe de travail (et de papotage ;)) avec Marie, Daniel et Valérie ! Quel plaisir de vous avoir rencontré et d'avoir passé ces 3 mois 1/2 de cours à vos côtés, même s'il fallait aller en cours tous les samedis :-) .

Étienne (Graouh ! On se redonne 3̶ 4 ans pour finir les jap' de la rue St Anne ?), Jonathan, Harris, Madeleine, Elie et Chryssa. Ces 2 années de Master m'ont permis d'apprendre à vous connaître et de passer de (courtes ?) vacances à la montagne avec vous. J'ai réalisé que vous étiez des amis sur lesquels je pouvais compter, en particulier pendant les nombreux projets chronophages rencontrés pendant ces 2 ans (qui a dit ISR ?).

Marie, Chachou, David, Sophie, Clémence, Jérémie, Lucie et Élise. Tout a commencé avec quelques plaques de chocolat et paquets de fraises tagada ramenés en cours d'algèbre générale et 8 ans plus tard, malgré un déménagement sur Paris, nous nous voyons toujours régulièrement. Les 2 années que j'ai passées à Lille sont de loin les meilleures de ma vie étudiante et cela grâce à vous.

Kam, Cam, Nicolas M., Nicolas L., Nicolas M., Pascaline, Arnaud et tous les autres que je ne cite pas car la liste serait trop longue ! Je ne compte plus le nombre d'années depuis que l'on se connait (cela date du primaire pour certains !), je suis toujours étonné que l'on arrive à se revoir régulièrement malgré les années qui passent. Merci à vous tous pour votre amitié et toutes ces années passées ensemble.

Boris Leroy qui après avoir sillonné pendant des années tout le nord-ouest de la France a atterri à 50 mètres de l'université ! Même si l'on ne se voyait pas souvent, c'était toujours une bonne occasion pour geeker et s'amuser (au hasard, se lever à 7H30 pour jouer à Bubble Bobble, nos premières soirées réseaux où l'on passait plus de temps à réunir des cartes réseaux, des câbles BNC et des bouchons plutôt qu'à jouer, se cacher dans le magasin de tes parents, en particulier au moment où je devais repartir chez moi, . . .). Merci pour ton amitié et tout ces moments passés ensemble.

Il m'était impossible de ne pas remercier tout le personnel médical des hôpitaux de Boulogne-sur-mer, Camiers, Arras, Necker et Foch, tous les médecins, kinésithérapeutes, infirmières, chirurgiens et pneumologues qui m'ont soigné depuis tout petit. Je n'aurai jamais pu en arriver là sans eux et cette thèse est aussi la leur.

Je tiens à remercier tout particulièrement ma mère, mon père et ma grand-mère pour leur soutien inconditionnel et pour m'avoir fait confiance malgré toutes ces années d'études (même si parfois, ce n'était pas évident à suivre :-)). De même, merci à Noémie, Marie et Valentine, mes 3 petites sœurs-cousines pour m'avoir accompagné pendant toutes ces années, pour tous ces repas du dimanche passés en famille à La Capelle, tout ces Noëls, en bref, pour tous ces moments passés ensemble.

Pour finir, je ne peux pas quantifier le soutien qu'a pu m'apporter Anaëlle pendant ses 4 années, pour sa gentillesse, sa bonne humeur et surtout pour m'avoir supporté (dans les 2 sens du terme !) pendant ces 4 années. Merci pour tout.

Contents

Chapter 1

Introduction

We live in a world of data and our daily production of data is growing exponentially. In 2013, the International Data Corporation (IDC) estimated the size of the Digital Universe at about 4.4 zettabytes (4.4×10^{21}) and forecasts that it will double every 2 years to reach 44 zettabytes by 2020 [48]. This trend is known as Big Data. Moreover, the number of connected objects in the Internet of Things, which refer to everyday life objects with an embedded connectivity to Internet giving them the ability to transmit data, was reckoned at about 20 billion of devices in 2013 and is expected to reach 32 billion by 2020, accounting for 10 % of the Digital Universe. These forecasts highlight the fact that our lives are surrounded by data and that this sustainable trend is going to increase in the following years.

Taking advantage of Big Data is a critical concern for industries like financial services, technology, healthcare, retailing or energy since it is considered to be one of the most important driver of value added for the future. Big Data can help firms to make better business intelligence decisions, for instance, how to understand their customer consumption habits, how to optimize their operational and monitoring processes, how to make better pricing decisions, and many more. Some example of applications using Big Data includes the Facebook Graph Search tool [19] for advanced multi-criteria search in their user graph which can answer complex queries for targeting the right customers or recommender systems as used for Youtube [25] or Netflix [58, 59] where the system recommends personalized sets of videos to users based on their activity on the website. However, it remains a challenge to be able to structure these data in a comprehensive way, to process them with a low latency, to extract the appropriate information efficiently, and to report these results to the final clients or decision makers.

Processing such tremendous amount of data raises important challenges for the system community. Typical Big Data applications such as large-scale analytical frameworks like MapReduce [26] or Spark [99], databases [40, 61] and web servers [7, 50] have important requirements in terms of computing and memory capabilities and where responsiveness and throughput are critical for a enjoyable user experience [76]. These programs are typically parallelized and rely on server class computer like multicore hardware located inside data centers as a computing platform. However, parallelisation is a notoriously difficult problem which can prevent leveraging the full computing power of such platforms, especially due to Amdahl's law [6, 36]. This law states that the potential speedup improvement obtained by parallelizing a program is limited

by the sequential portion of this program. Typical sequential portions of a program are called critical sections: they protect shared data from multiple concurrent accesses and are surrounded by locks to ensure consistency.

However, because of the complexity of these applications, some critical sections may not be efficient in all execution configurations. Such critical sections can impede thread progress under specific conditions, which can drastically degrade the time for the server to process requests. Developers generally try to find these badly written critical sections during the performance evaluation phase but it is not always enough to find all of them. There is 3 main reasons to this situation:

– Difficulty to reproduce a real execution environment: the software will most likely use a representative dataset of the expected workload for testing. In the best case, developers will try to simulate the most representative dataset, close as much as possible to real world conditions, to stress the application. However, this dataset is dependent from the core business and will be populated by users, thus making it completely unknown prior to deployment. It could be far from what developers are expecting, in particular if the software is flexible enough to be used in a wide variety of situations,

– Difficulty to simulate every possible scenarios of execution: the testing workload applied to the software is generally composed of a mix of predefined set of queries. Users queries are not predictable in advance and they exposed the software to a wide variety of queries to process. It is difficult to find out how to stress the software with a workload close to the one it will experience under practical conditions of use,

– Impossible to test every hardware configuration: developers usually have access to a restricted set of machines for testing. They can't evaluate applications on a wide set of architecture and processors where results could vary. Performance can also greatly vary between different versions of the same operating system or JVM. Sometimes, they also carry out their tests on their own development machine which is far from a typical server class computer, made of an important number of cores and memory size. It is not practical to test all of these combinations and developers end up testing only a subset of architecture, operating systems, virtual machine, and application version configurations.

For these reasons and despite a thorough testing, it is difficult to simulate all scenarios exhaustively. Therefore, throughput and responsiveness may be hampered in situations that were not expected by developers during the development phase and that will be discovered while it is deployed in real world conditions, with drastic effects on the customers experience. For instance, Google's CEO Marissa Meyers reported at Google I/O conference that an increase in latency of half a second could lead to a drop of the traffic by 20% [66], leading to less advertising revenues.

Java is regularly used to implement these complex multithreaded applications. It has become one of the most used programming languages thanks to its safety, flexibility, and mature development environment [91]. Nevertheless, the Java language is notoriously ill-adapted to multicore architectures. The main concurrency abstraction provided by Java is the *synchronized* keyword, which encourages the use of coarse-grained synchronization. Despite efforts made by the Java community with, for instance, the java.util.concurrent package [62] which aims to offer

a finer set of abstractions for concurrency control, synchronized blocks are still widely used. For instance, there is approximately 7410 synchronized blocks located in the Java Class Library of Java 7. Applications cannot be fine-tuned for execution on a specific multicore configuration, taking into account, e.g., cache behavior or memory hierarchy, because such features are hidden by the Java Virtual Machine (JVM). Finally, the training and experience of Java developers are typically more oriented towards aspects of high-level software structuring, and less towards low-level synchronization issues.

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 [61][1]. Under a specific setting, with three Cassandra nodes and a replication factor of three, when a node crashes, the latency of Cassandra is multiplied 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.

The research conducted in this thesis investigates the topic of lock profiling. More precisely, we focused on the problem of performance degradation of server-class applications due to lock contention, with an emphasis on Java applications running on top of multicore architectures. For these reasons we have previously highlighted, we have designed a lock profiler with the following properties:

1. The profiler must use a metric that indicates whether a lock impedes thread progress. The profiling report for the developers must give a clear insight about the impact that locks have on application performance, especially in terms of responsiveness and throughput. This will allow the developer to concentrate his debugging effort on a bug that does really hamper the application performance,

2. The profiler should recompute this metric periodically, to be sensitive to the different

---

[1] https://issues.apache.org/jira/browse/CASSANDRA-3386.
[2] http://wiki.apache.org/cassandra/HintedHandoff.

phases of the application. Complex applications servers are stressed by various factors like an unpredictable environment, several peaks loads at different time of the day, different mix of queries, and users behavior, all of that being not predictable theoretically. All these scenarios cannot be foreseen in a testing environment and thus it is not possible to detect every lock contention problem. A profiler computing and reporting regularly a lock contention metric will find issues related to users and environment characteristics,

3. The profiler must incur little overhead in order to be used *in-vivo*. There is a need for an *in-vivo* profiler, i.e. a profiler that continuously monitors the application while it is running, but users will not be willing to use a profiler that degrades drastically their application behavior. Intuitively, it is also contradictory to slow down an application continuously for the purpose of finding a potential bug that may hamper application performance. Therefore, the overhead impact of the profiler must be as limited as possible in order to be not noticeable by the end user.

In this thesis, we propose a new lock profiler, called Free Lunch, designed around a new contention metric, *critical section pressure* (CSP). This metric aims to evaluate the impact of lock contention on overall thread progress. CSP is defined as the percentage of time spent by the application threads blocked while acquiring locks during a time interval. This indicates the percentage of time where threads are unable to make progress, and thus the potential loss in performance. Free Lunch is especially targeted towards identifying *phases* of high CSP *in-vivo*: the application is sampled continually over several time intervals during which CSP is computed for each lock. 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 [38].

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 JVM by extending them with an additional data structure containing profiling data. These lock structures are already thread-safe and thus Free Lunch does not require any additional synchronization to access 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. Free Lunch also relies on hardware specific instruction that provide efficient time management facility, allowing for minimal instrumentation of the code in charge of locking. 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 [89]. 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 key contributions are as follows:

- Theoretically and experimentally, we have found that the lock contention metrics used by the existing Java lock profilers HPROF [42], JProfiler [52], Yourkit [97], MSDK [69], IBM

Health Center [41], Java Lock Monitor [67] and Java Lock Analyzer [49] 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 has remained undetected to Cassandra developers because it is triggered under a specific scenario and only arise during a short 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 to verify that the locking behavior either does not hamper enough thread progress to have a significant impact on application performance or could not easily be improved.

- On the DaCapo 9.12 benchmark suite [12], the SPECjvm2008 benchmark suite [87] and the SPECjbb2005 benchmark [86], we found 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 [42], JProfiler [52] and Yourkit [97], on the same set of benchmarks incur a performance overhead of up to 4 times, 7 times and 1980 times, respectively, making them unacceptable for *in-vivo* profiling.

Organization of the document.

The rest of this thesis is organized as follows:

- Chapter 2 introduces locking mechanisms available in Java, the seven lock profilers presented in this thesis for the evaluation and the state-of-the-art of profiling for parallel applications. It presents locking features provided by the Java language and its implementation in a JVM, focusing in particular on the locking subsystem of the OpenJDK Hotspot JVM [89]. The seven lock profilers evaluated with Free Lunch are presented in details. The state-of-the-art focuses on lock profilers found in the literature and profilers solving other performance issues in the context of concurrent applications on multicore hardware.

- Chapter 3 presents the main contributions of the research work presented in this thesis, namely, a study of the effectiveness of existing metrics at finding lock performance issues, the Critical Section Pressure metric (CSP) and the Free Lunch profiler. The effectiveness at finding lock contention issues of lock metrics found in the seven lock profilers are evaluated against typical synchronization scenarios found in multithreaded applications. Based on observations made in this study, we design the CSP metric, a new metric that assesses the impact of lock contention on overall thread progress and its implementation in Free Lunch, a lock profiler embedded inside Hotspot. Free Lunch profiles Java applications and reports the CSP continuously during the whole execution while achieving a low overhead.

- Chapter 4 presents the evaluation over a set of benchmarks. First, it includes 2 experiments about overhead: a comparative study of the overhead of Free Lunch versus state-of-the-art Java lock profilers presented in Section 2.2 and a detailed evaluation of the overhead costs involved by each feature of Free Lunch. Then, a set of microbenchmarks compares state-of-the-art Java lock profiler metrics with CSP and a detailed analysis of lock CSP over a restricted set of applications is presented, including experiments about how CSP is effective at finding lock contention issues. Finally, a real use case highlights a performance bug found in Cassandra [61], a scalable and highly available distributed database, proving that Free Lunch is effective at finding short-lived and critical lock contention issues.

- Finally, Chapter 5 concludes this thesis and draws future perspectives.

Chapter 2

Background

This chapter describes locking mechanisms in Java, the seven lock profilers analyzed in this thesis, and the state-of-the-art in profiling for concurrent applications. Section 2.1 introduces locking features provided by the Java language and its implementation in a JVM with an emphasis on the locking subsystem of the OpenJDK Hotspot JVM [89]. Section 2.2 presents in detail the lock profilers HPROF, JProfiler, Yourkit, Multicore Software Development Kit, Health Center, Java Lock Monitor, and Java Lock Analyzer used in this thesis for the evaluation in Chapter 4. Section 2.3 describes recent research done in the field of profiling locks and concurrent applications on multicore hardware and distributed systems.

2.1 Locking in the Hotspot 7 JVM

The Java language provides *synchronized* statements and methods ensuring mutual exclusion [39]. Synchronization is done by the way of Java objects, acting as synchronization proxies and being the parameters of synchronized statements. When speaking of Java locks, the words 'Java object' and 'lock' can be used interchangeably since they refer to the same data structure. Java locks are re-entrants and can be locked recursively by the lock holder.

2.1.1 Lock data structure

In Java, every object can potentially be used as a lock but only a minority will be used as such, therefore it is important that synchronization data remains small in order to not waste memory space. Therefore, the data managing the locking status of the object is directly embedded in the Java object header. The Figure 2.1 presents the header of a Java object which size is of two machine word. The first word, called *mark word*, contains information related to synchronization. The second word is a pointer to the data structure representing the object's class.

There is three different lock algorithms implemented in the Hotspot virtual machine:

- Biased locking (or lock reservation): this algorithm comes from the observation that the majority of Java objects are locked by at most one thread during their lifetime, therefore the lock is never locked by more than one thread. A lock reserved for a thread does not require a write to acquire it,
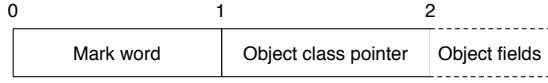
Figure 2.1: Java object header.

Lock algorithm	Lock state	Mark Word	
		Field	Tag
Biased Locking	Free	0	101
	Locked	Thread ID	101
Stack-locking	Free	0	01
	Locked	Pointer to the Lock record	00
Monitor		Pointer to the Monitor	10
		Reserved	11

Figure 2.2: States of the mark word.

(a) State before acquiring the lock.

(b) State after acquiring the lock.
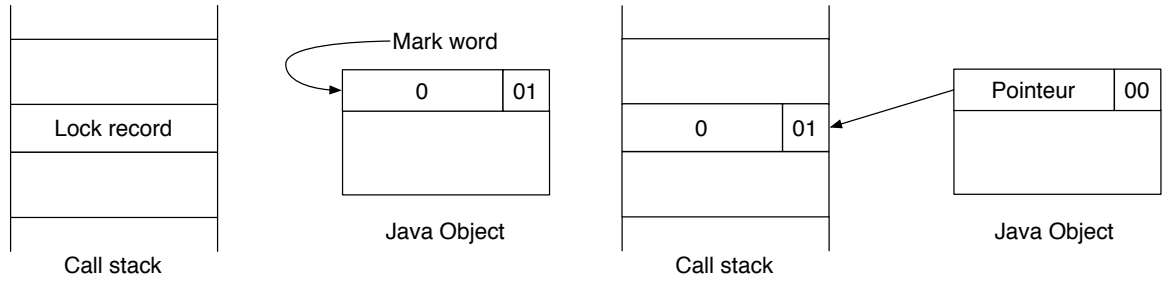
Figure 2.3: Acquiring a lock with the Stack-locking algorithm.

- Stack-locking: this algorithm is used when the lock is shared between several threads but there is no thread attempting to lock it when it is already held by another one,

- Monitor algorithm: this algorithm is used when several threads try to acquire the same lock concurrently. This algorithm manages threads blocking while waiting for the lock to be released.

Algorithm	Sharing	Contention
Biased locking	Not shared	Not contended
Stack-locking	Shared	Not contended
Monitor algorithm	Shared	Contended

Table 2.1: Use case of lock algorithms in Hotspot.

These lock algorithms are summarized in Table 2.1. The same implementation strategy is used in other modern JVMs, such as Jikes RVM [3] and VMKit [37]. These three different lock algorithms need a way to distinguish which one is currently in use. This is the role of the mark word (see Figure 2.2). The mark word allows the JVM to know the locking status of the object by associating the lock algorithm in use and the state of the lock (locked or unlocked) with the *tag* field.

The present design of the mark word rests on locks such as thin lock [8] and tasuki lock [74] which have 2 states: the flat state and the inflated state. The flat state is used when one

Figure 2.4: State-transition diagram of the mark word between lock algorithms.

thread holds the lock or at most one thread tries to acquire it. In this case, there is no other thread waiting to acquire the lock and the mark word data structure is enough to deal with synchronization. The lock switches to the inflated state when several threads try to acquire it concurrently. A data structure, called *monitor*, more costly in terms of memory space, is then linked to the object and acts as a surrogate for locking. The moni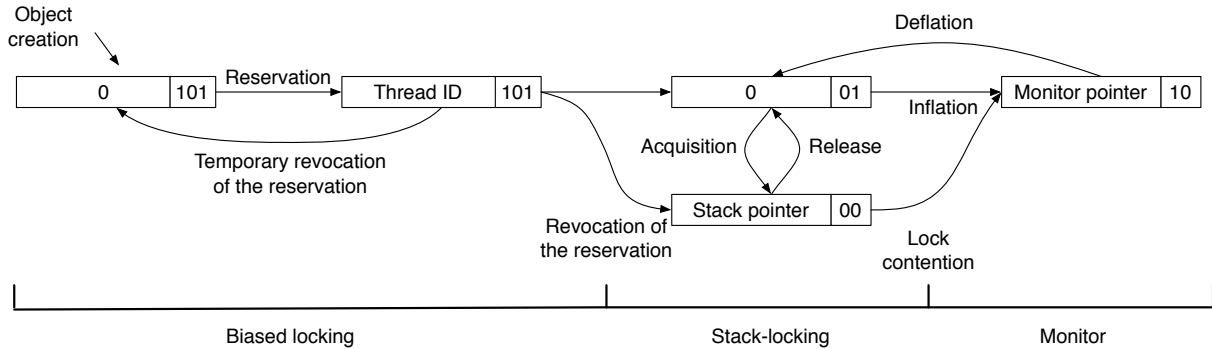tor helps to deal with access contention to the lock by managing a queue where threads block while waiting for the lock to be released.

Figure 2.4 presents the state-transition diagram between the three locking algorithms which are described in details in the following sections.

2.1.2 Biased locking

Biased locking (or lock reservation) [30] is used when a lock is acquired by at most one thread, meaning that it is not locked between different threads. This mechanism relies on the observation that most of Java locks are locked by at most one thread during their lifetime and are never shared with other threads. The idea is to reserve the lock for a particular thread as long as there is no other one trying to acquire it, otherwise the reservation is canceled. The reservation is done with an atomic Compare-and-Swap (CAS) but all subsequent acquisitions only need one read to ensure that the reservation is still valid. The Hotspot 7 JVM implements a costly lock reservation revocation protocol since all Java threads must be stopped for that purpose. Other algorithms that does not require to stop all Java threads exist in the literature [57, 75, 80, 94] but they are not implemented in Hotspot.

Acquiring the lock reservation

A newly allocated object uses by default the biased locking algorithm. The object is anonymously reserved: no other thread has yet reserved it and any one can try to do so. The thread sets atomically its identifier in the object mark word to acquire the reservation. The thread has only to ensure that it is still owning the lock reservation for subsequent acquisitions. This is done by checking that the lock is still using biased locking and by checking with a single read that the thread identifier of the mark word still match its own identifier. The lock reservation holds as

long as there is no other thread trying to acquire the lock.

Switching from Biased locking to Stack-locking

The thread owning the lock reservation remains the owner of the lock as long as no other thread tries to acquire it or as long as it does not release the lock reservation by itself. Another thread trying to acquire the lock triggers the lock reservation revocation and makes the lock switches to the Stack-locking algorithm. The revocation changes the state of the mark word to indicate that it is now using the Stack-locking algorithm as it is shown in Figure 2.4.

During the revocation, it is possible that a thread is executing a critical section protected by a lock for which it holds the reservation. In this case, this lock must remain locked by the same thread when it is switched to the Stack-locking mechanism, and this thread will use the corresponding release mechanism for unlocking. In order to know if the a thread is inside a CS, the Hotspot JVM must stop all Java threads which is a costly operation since no threads run anymore meanwhile. There are some heuristics in Hotspot [82] to disable temporarily or definitively biased locking for a particular Java class if their objects often experience revocation.

2.1.3 Stack-locking

Stack-locking is used when several threads lock the same lock in turns: the lock is shared between them but there is no attempt to lock it while it is already held. Under these conditions, an atomic CAS is used to modify the ownership of the lock since this instruction is not costly when there is no contention. A situation of lock contention occurs when a thread tries to acquire a lock while it is already held by another thread, thus the lock switches to the Monitor algorithm.

Acquire a lock

The lock holder has to ensure 2 things: (i) to indicate to other threads that the lock is owned, and (ii) to be able to know quickly if a thread owns a lock or not. For that purpose, a memory space, called *lock record*, is always allocated for each possible synchronized statement inside a method on the thread's call stack (cf. Figure 2.3a). The thread will store the current mark word inside the lock record when it acquires the lock (cf. Figure 2.3b). This mechanism allows a thread to know if it owns a lock by verifying that the pointer inside the mark word belongs to the range of memory address of its own call stack.

Java locks are re-entrants and thus can be acquired recursively. The first locking must be differentiated of the subsequent ones to ensure to release the lock only for the last unlocking and not during a recursive one. The thread knows it is a recursive locking when the mark word already contains a pointer to a lock record in its own call stack, which means that it already owns the lock. In this case, the thread sets the lock record value to NULL, thus avoiding to maintain a costly external counter counting the number of recursive acquisitions by checking instead that the pointer inside the mark word belongs to the range of memory address of the thread's call stack.

Release a lock

The release of the lock is done in a symmetric way. The thread first checks that it is the owner of the lock and throws an IllegalMonitorStateException if it is not the case. Then, if the value of the lock record is NULL, it is a recursive release and the thread does not perform any action. Otherwise, the lock record contains the value of mark word during the first acquisition: the lock must be definitively released. The value of the mark word contained in the lock record is replaced atomically in the object mark word. The lock is then free and can be acquired by another thread.

Switching to the Monitor algorithm

The lock switches to the Monitor algorithm when a thread tries to acquire a lock while it is already locked by another thread. The thread wanting to acquire the lock will try to transform it from the flat mode to the inflated mode, thus changing the algorithm currently in use to the Monitor algorithm. Consequently, the thread will use the release mechanism of the Monitor algorithm for unlocking. The exact process is described in details in the next section about the Monitor algorithm.

2.1.4 Monitor algorithm

The Monitor algorithm [29] is used when there is contention: several threads try to acquire the same lock concurrently (at least one attempt to acquire the lock is done while it is already held).

Having multiple threads trying to acquire the same lock simultaneously raises multiple challenges. It is useless to let threads try to acquire the lock by spinning for an extensive period of time, thus wasting CPU time and forbidding other threads to make progress. Therefore, there is a need for a mechanism to stop threads at some point and to make them wait for the lock to be released. However, the mark word design can not handle such mechanism by itself due to its small size. A bigger data structure is required and a way to associate it to the original lock.

Monitor data structure

To address these issues, a data structure, called *monitor*, is associated to the Java object. The object stores a pointer to this data structure in the mark word as shown in Figure 2.2 and reciprocally, the monitor stores a pointer to this object in the **_object** variable. This data structure contains all locking data for synchronization purposes when using the monitor algorithm. Therefore, this requires a variable to keep the lock owner of the lock (**_owner**) and the number of lock recursions (**_recursions**) in this data structure since there is no space left in the mark word for these information.

The monitor manages threads stopped due to the lock already owned by a thread and waiting for it to be released. This is the role of the **_EntryList** linked list that contains threads currently in this state. The monitor is also in charge to wake up threads after the lock is released. There is not point to wake up threads when there is already some threads trying to acquire the lock. In order to detect this situation, the monitor relies on the **_succ** variable which is set regularly by threads actively trying to acquire the lock.

The link between the monitor and the Java object is deleted when lock contention is over, the lock switches back to Stack-locking and the monitor structure is recycled for a further usage.

Here is a summary of the monitor data structure we have presented so far:

- **_owner**: a variable containing the thread ID of the thread owning the lock or NULL if the lock is unlocked. It is always set by using an atomic CAS,

- **_recursions**: a variable keeping the number of recursions done by the lock owner when it acquires the lock recursively,

- **_EntryList**: a linked list of blocked threads because the lock was already owned waiting for it to be released,

- **_object**: a pointer to the Java object associated with the lock,

- **_succ**: a variable containing the identifier of one of the threads trying to acquire the lock or NULL if there is no threads spinning in order to acquire the lock (either no thread wants the lock or all threads are already blocked after trying to acquire it for some time).

Switching from Stack-locking to the Monitor algorithm

Inflation consists in modifying the lock data structure to use the Monitor algorithm. This mechanism is triggered when a thread tries to acquire a lock using the Stack-locking algorithm while it is already locked by another thread.

The thread failing to obtain the lock will be in charge of switching from the Stack-locking algorithm to the Monitor algorithm. This is a 3-step process. Firstly, the thread allocates an empty monitor data structure and initializes its fields. Secondly, the thread will attempt to set atomically the mark word with the special value INFLATING to notify all other threads that it is currently initializing a monitor inside the mark word. Lastly, the thread will replace the value INFLATING of the mark word by the monitor memory address. The thread can fail to install the INFLATING value to the mark word if the owner of the lock releases it meanwhile or if another thread tries to inflate the lock at the same time. In this case, the thread tries again from the start the procedure.

Acquire a monitor

The implementation of the Monitor algorithm has a fast locking acquisition mechanism called *fast-path* and a slower one called *slow-path*. The fast-path manages simple lock acquisition cases and is written in assembly for a quick execution. The slow-path takes care of more complex lock acquisitions. It is called when the fast-path fails or directly when a complex locking case is detected early enough. A monitor is acquired by setting the thread identifier atomically into the **_owner** variable.

The fast-path permits a quick lock acquisition when the lock is free. It solely consists in setting the **_owner** variable with the thread identifier. The acquisition can fail due to 2 reasons. In the first case, the thread already owns the lock, therefore it simply increments the recursions

variable and resumes its execution. In the second case, another thread already holds the lock, thus the thread will use the slow-path to acquire it.

The slow-path ensures that lock acquisition remains fast even if several threads are contending to acquire the lock. Previous work has shown that multiple atomic operations done simultaneously by many cores on the same cache line trigger cache entries invalidation and thus can saturate the memory interconnect and slow down the application [13]. Therefore, threads taking the slow-path do some *backoff-spinning*. Backoff-spinning is a technique that consists in actively trying to acquire the lock while progressively increasing the time between 2 attempts [1]. This limits contention on the interconnect by reducing the number of simultaneous requests on the lock cache line. If the thread did not manage to acquire the lock after some time, it stops spinning, blocks, and is enqueued in the **_EntryList** queue. It remains blocked until the lock is finally released. Once the thread stops blocking, it resumes its execution and starts the lock acquisition process back from the start.

The thread also sets regularly his identifier into the **_succ** variable during backoff-spinning to indicate that it is actively trying to acquire the lock. This is done for the purpose of releasing the lock efficiently and is explained in the following paragraph.

Release a monitor

The release of the monitor is based on a mechanism called competitive handoff. This mechanism ensures that the thread which releases the lock checks that at least if one thread is actively trying to acquire it, otherwise it wakes up only 1 blocked thread waiting to acquire it. This avoids to wake up a blocked thread if there is already some of them trying to acquire the lock. Therefore, all blocked threads are not woken up uselessly and do not wast resources.

Before releasing the lock, the thread checks that it is the owner by comparing its identifier with the **_owner** variable, raising a *IllegalMonitorStateException* in the opposite case. It also checks the value of **_recursions** to ensure it is not a recursive exit, in this case, it just decrements the value, remains the lock owner, and continues its execution.

In all other cases, the thread releases the lock definitively by setting the **_owner** variable to NULL. The thread is then facing 3 different cases depending on the value of the **_EntryList** (which contains all threads blocked waiting for the lock to be released) and **_succ** (which indicates that at least one thread is trying actively to acquire the lock):

- **_EntryList** is empty and **_succ** is NULL: there is neither blocked threads waiting to acquire the lock nor threads actively trying to acquire it, thus the thread does not have to do anything,

- **_succ** is not NULL: at least one thread is actively trying to acquire the lock, thus there is no need to wake up a thread since the active one will end up acquiring the lock,

- **_EntryList** is not empty and **_succ** is NULL: there is at least one blocked thread waiting for the lock to be released and no thread trying actively to acquire it. The thread will wake up a thread in the **_EntryList** queue to let it acquire the lock.

Deflating a monitor/Switching back to Stack-locking

Lock contention can be temporary, therefore it is appropriate to reuse the Stack-locking algorithm at some point because it is more CPU and memory efficient than the Monitor algorithm. This mechanism is called *deflation* and consists in removing the link between the monitor and the Java object and to revert the lock to the Stack-locking algorithm. It is difficult to know exactly when lock contention is really finished, thus this operation is done speculatively and regularly by the JVM.

In Hotspot, this operation is done during a Safepoint [83], a specific operation where all Java threads are stopped for operations like garbage collection or revocation of the lock reservation. The lock algorithm is reversed back to the Stack-Locking algorithm if the monitor is not currently locked and if no thread is trying to acquire it. The lock will be inflated again and a fresh monitor will be associated to the Java object if lock contention is again encountered after deflation.

2.2 Lock profilers

This section present the state-of-the part lock profilers used in this thesis for evaluating lock contention metrics in Section 3.1 and for the evaluation in Chapter 4. Three of them are designed for the Hotspot 7 JVM (HPROF [42], JProfiler [52] and Yourkit [97]) and four for the IBM J9 JVM, (Health Center [41], Multicore Software Development Kit [69], Java Lock Monitor from the Performance Inspector suite [67] and Java Lock Analyzer [49]).

These profilers make a wide use of the JVMTI interface [55], a set of hooks which permits the inspection of the internal state and data structures of the JVM. It was developed to replace the previous JVMPI [54] and JVMDI [54] interfaces. Every profiler suitable for Hotspot use the JVMTI interface [55] to obtain information from the JVM. On the contrary, profilers suitable for the IBM J9 JVM are using either the JVMTI interface or a native interface specific to IBM like in [45] whose specification is not disclosed publicly. Each profiler presentation includes a detailed output of a profiling session of Xalan from the DaCapo 9.12 benchmark suite [12] except for Java Lock Monitor and Java Lock Analyzer because we were not able to run them on our platforms since they are not maintained anymore.

2.2.1 HProf

HProf [42] is an open-source legacy profiler designed by Sun Microsystems and shipped with many JVM like Hotspot and J9. Releases of J2SE from 1.2 (December 1998) through 1.4 contained a version of HPROF built on the JVMPI interface. The newer JVMTI in J2SE 5.0 (September 2004) replaced both JVMDI [53] and JVMPI, therefore HProf has been fully rewritten for compliance with this new interface. HProf is provided as a dynamically-linked shared library.

HProf is able to profile many features of the JVM: heap object allocation by class, CPU usage (by sampling or by counting time spent and number of entries in a method) by class, and lock usage. Each allocation or CPU site is associated to a stack trace showing calling-context-sensitive profiles associated with the observed behaviour. All profiling features rely on

the JVMTI interface but heap and CPU profiling need additional Byte Code Insertion [10, 11] to work.

Using HProf for lock monitoring

The HProf agent is directly embedded in the JVM and is started for lock profiling with the command line $ java −agentlib:hprof=monitor=y java_program. The option monitor disables every other profiling functionality and only enables lock profiling. The profiler report is written in a human-readable format to the java.hprof.txt file.

The Output 2.1 shows a typical output generated by HProf where some parts have been shrunk for clarity.

The first part (line 2 to 5) presents a timeline about thread creation and destruction and about each waiting operation. The creation (line 2) and the destruction (line 5) of each thread is presented along with its thread ID, thread name, and the group thread to which it belongs. This part also contains every call to the wait() method. A notification is written to the file (line 3) each time a thread calls wait(), with the class of the object acting as the synchronization object, the timeout if the thread has to wait until a specified amount of time has elapsed, and the thread ID of the thread waiting on the object. Once a thread finished to wait, a second notification is written (line 4) with the class of the object acting as the synchronization object, the time during which the thread has waited and the thread ID of the waked up thread.

The second part (line 7 to 17) presents a series of trace entries. A trace entry is made of a Trace number and a stack trace. The Trace number helps to link the results reported in the third part of the output to a trace entry. The stack trace contains a set of active stack frames (by default 4 frames) referring to the code path of the application at that particular time. These information are used later in the third part to link lock statistics to stack traces.

The third part (line 19 to 31) shows the summary of locks statistics about the profiled application. The last part allows us to compute the Acquiring time of a lock divided by the Acquiring time of all locks and the Acquiring time of a lock divided by the Elapsed time metrics. The first line (line 19) displays the amount of time spent in locking operations by all threads with the current date. Right after is a list of every monitor, ranked by the percentage of time spent while acquiring a lock divided by the total acquiring time of all locks. Additional information are presented like the accumulated percentage of time spent acquiring a lock (from the first the last monitor), the number of time the lock was locked, the Trace number to which the lock is linked in the second part of the output, and finally the Java class of the lock.

Output 2.1: Profiling output of HPROF

```
1  ...
2  THREAD START (obj=50000472, id = 200021, name=''Thread−18'', group=''main'')
3  WAIT: MONITOR Lorg/dacapo/xalan/XSLTBench$WorkQueue;, timeout=0, thread 200021
4  WAITED: MONITOR Lorg/dacapo/xalan/XSLTBench$WorkQueue;, time_waited=159, thread 200021
5  THREAD END (id = 200021)
6  ...
7  TRACE 300254:
8    org.apache.xml.utils.XMLReaderManager.getXMLReader(XMLReaderManager.java:84)
9    org.apache.xml.dtm.ref.DTMManagerDefault.getXMLReader(DTMManagerDefault.java:610)
10   org.apache.xml.dtm.ref.DTMManagerDefault.getDTM(DTMManagerDefault.java:282)
11   org.apache.xalan.transformer.TransformerImpl.transform(TransformerImpl.java:699)
```

```
12 ...
13 TRACE 300271:
14    java.util.Hashtable.get(Hashtable.java:433)
15    org.apache.xalan.templates.TemplateList.getTemplateFast(TemplateList.java:508)
16    org.apache.xalan.templates.ElemApplyTemplates.transformSelectedNodes(ElemApplyTemplates
        ↪ .java:296)
17    org.apache.xalan.templates.ElemApplyTemplates.execute(ElemApplyTemplates.java:178)
18 ...
19 MONITOR TIME BEGIN (total = 1 ms) Tue Mar 17 17:12:49 2015
20 rank     self    accum     count trace monitor
21    1 98.47% 98.47%   549035 300271 java.util.Hashtable (Java)
22    2  0.40% 98.87%       63 300254 org.apache.xml.utils.XMLReaderManager (Java)
23    3  0.24% 99.11%       56 300267 org.apache.xml.utils.XMLReaderManager (Java)
24    4  0.08% 99.19%       19 300224 org.dacapo.harness.DacapoClassLoader (Java)
25    5  0.07% 99.26%      780 300237 java.util.Properties (Java)
26    6  0.07% 99.32%      262 300279 java.util.Properties (Java)
27    7  0.07% 99.39%      987 300275 org.apache.xpath.axes.IteratorPool (Java)
28    8  0.04% 99.43%       47 300180 org.dacapo.harness.DacapoClassLoader (Java)
29    9  0.04% 99.47%      233 300242 java.util.Properties (Java)
30   10  0.04% 99.51%      315 300241 java.util.Properties (Java)
31   11  0.03% 99.54%      613 300305 sun.net.www.protocol.jar.JarFileFactory (Java)
32    ...
33 MONITOR TIME END
```

2.2.2 JProfiler

JProfiler [52] is a general-purpose, commercial and closed-source profiler for Java developed by EJ-Technologies since 2001. JProfiler provides a dynamically-linked shared library to link with the JVM on startup for collecting data, and a user-friendly graphical interface for controlling the profiling process and analyzing results harvested by the library. There is 3 mode of usage for the developer: he can use it remotely by establishing a connection with the server and run the GUI on its own laptop, he can profile an application an application locally but the overhead of using the GUI on the same computer could bias the results, and finally he can run the application in headless mode, which means that the JProfiler library will log all events to a file and export it for a later usage with the GUI. JProfiler also provides addons for analyzing profiling data that can be integrated into several IDEs like Eclipse [33] or IntelliJ [47].

JProfiler supports CPU, memory profiling, heap memory inspection, thread profiling, garbage collector profiling, JVM telemetry and monitor profiling. It also gives the ability to setup dedicated probes for databases or specific Java frameworks to profile them more efficiently. Locks from the java.util.concurrent package [62] can also be proifled by JProfiler.

Using JProfiler for lock monitoring

JProfiler is started with the command line $ java −agentpath:~/jprofiler7/bin/linux−x64/libjprofilerti. so java_program. JProfiler enables many profiling features by default. These features need to be disable to avoid profiling useless data that are not related to locking, and thus could increase the application overhead induced by the profiling. This is done by configuring the session settings with the graphical interface.

A new session is first created by starting JProfiler, then clicking on Start center, then on the New Session tab and finally by clicking on New session. Figure 2.5 shows the newly opened

Session settings menu with the Application settings tab. The user has to specify a session name,the server IP and port for the remote connection in this tab. The Profiling settings tab presented in the Figure 2.6 permits settings customization such as method call recording or CPU time measurement. We disable all possible parameters (except for locking) in order to lower the overhead as much as possible. An estimation of the expected overhead is presented in the Performance section based on these parameters. The last configuration panel is the Triggers settings tab presented in Figure 2.7 where the user can create triggers, a defined set of actions launched when a specific event happens. We create a trigger, launched at the startup of the JVM, to disable recording, call tracer, probe recording, and probe tracking and also to start monitor recording. This way, locking events are recorded at the very beginning of the application. After setting all these parameters, the Session startup tab shown in Figure 2.8 summarizes the choices of the user before the application starts.
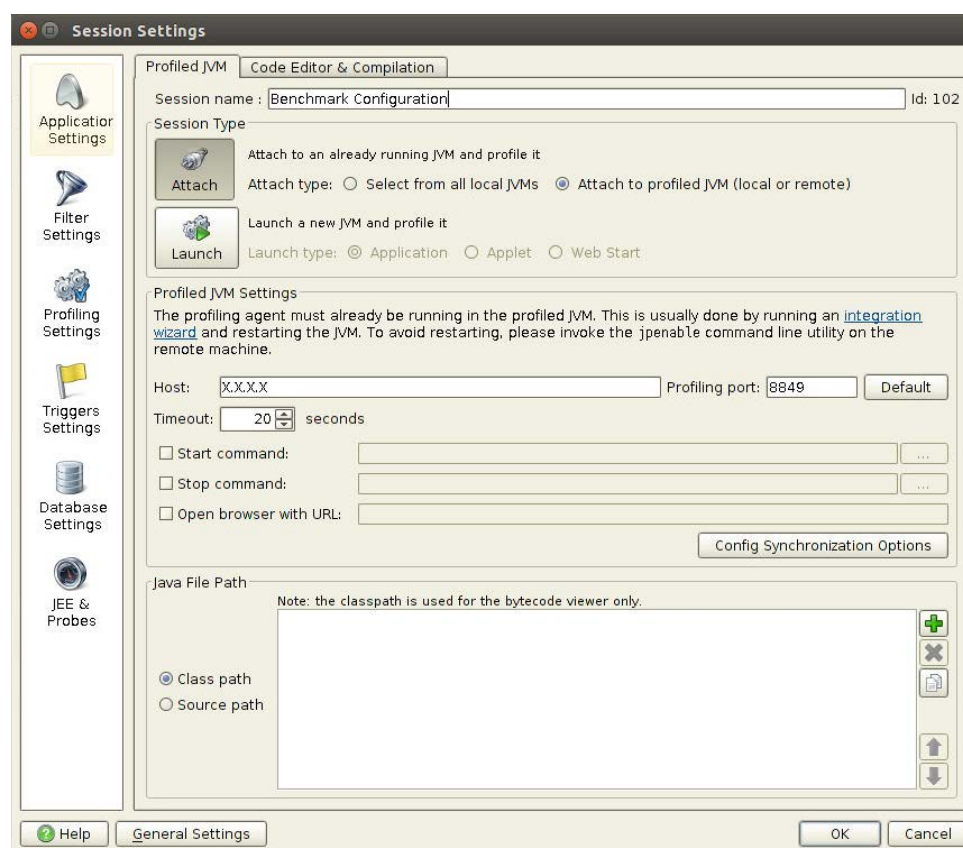
Figure 2.5: JProfiler Session settings tab.

The monitor & locks profiling view provides several information about the locking behavior of the application. This tab presents the block count (number of times a lock is locked when a thread tries to acquire it), the block duration (the total duration during when threads have blocked while trying to acquire the lock), the wait count (the number of times the wait() method was called on the monitor), and the wait duration (the total wait duration for all threads). In particular, the Monitor Usage Statistics tab allows us to compute the Acquiring time of a lock
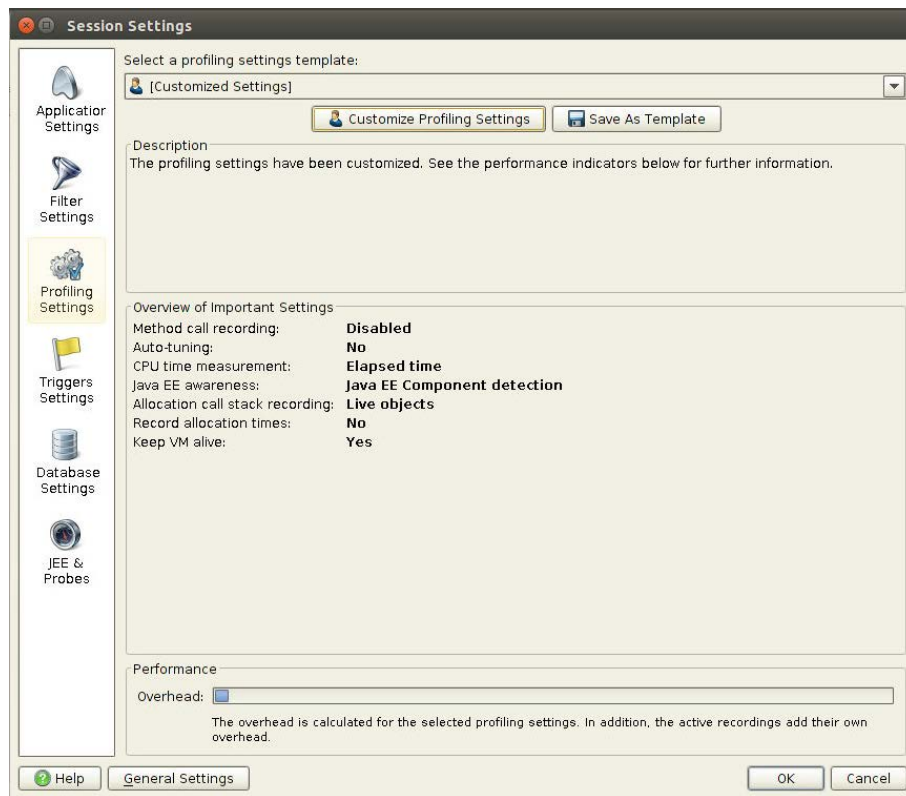
Figure 2.6: JProfiler Profiling settings tab.

divided by Elapsed time metric. These information are grouped either by monitors (Figure 2.9), by class of monitors (Figure 2.10), or by threads (Figure 2.11).

Views about the locking behavior summarizing additional data are available:

- Current Locking Graph: this view shows graphically monitors that are currently involved in a waiting or blocking operation,

- Current Monitors: this view shows monitors that are currently involved in a waiting or blocking operation with detailed statistics about the locking duration, the object class, the owning and waiting thread, and the monitor class and ID, as shown in Figure 2.12,

- Locking History Graph: this view visualizes the recorded locking situations in the JVM by navigating through all locking events,

- Monitor History: this view shows the sequence of waiting and blocking operations on monitors.
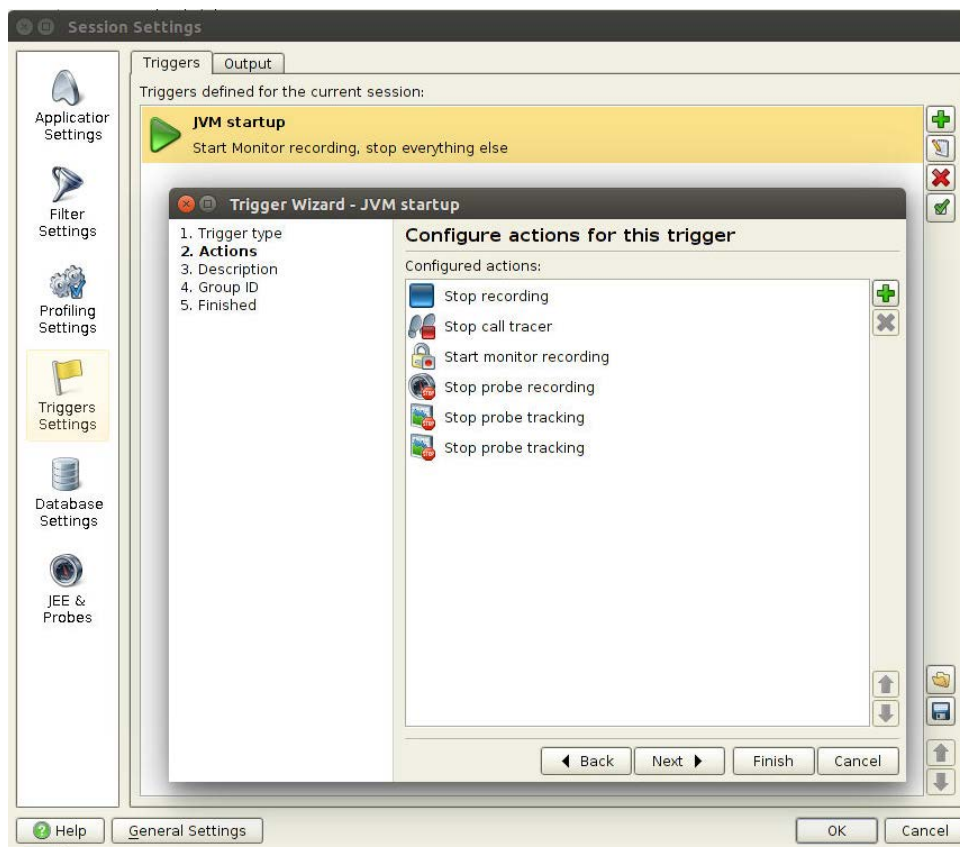
Figure 2.7: JProfiler Triggers settings tab.
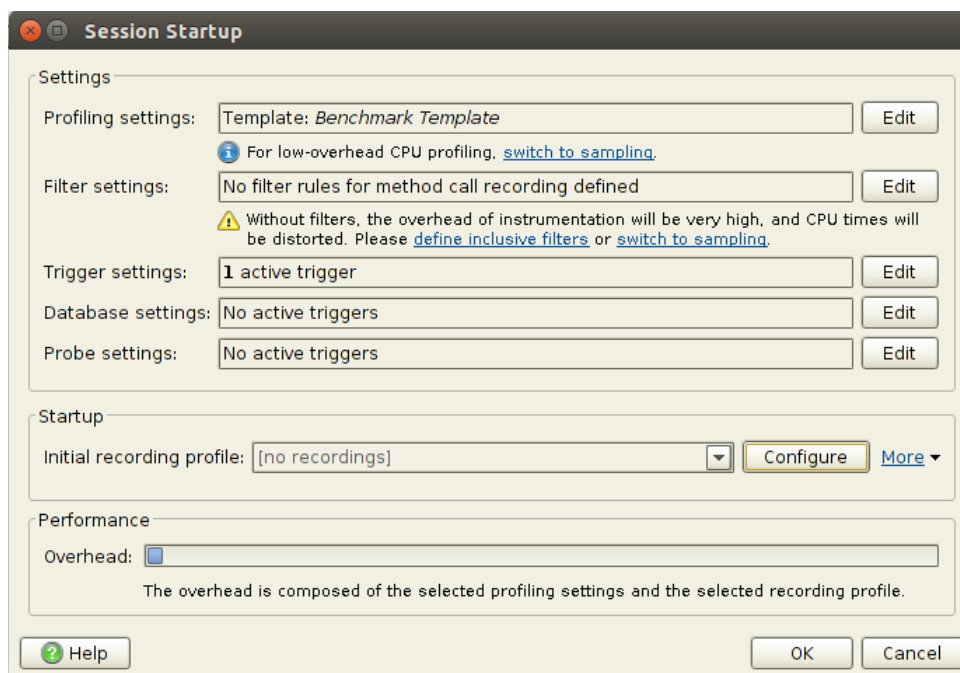
Figure 2.8: JProfiler Session startup tab.
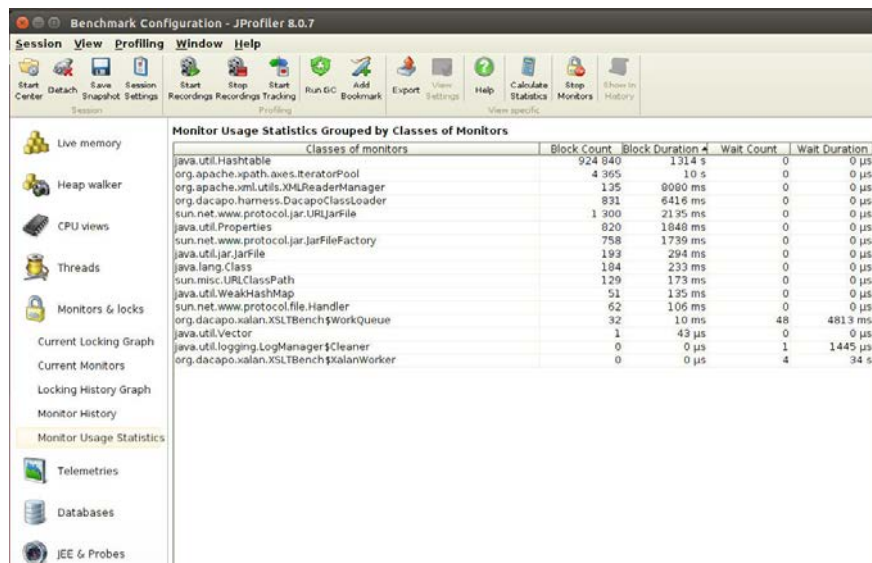
Figure 2.9: View of Monitor Usage Statistics, grouped by class of monitors, sorted by blocking duration.
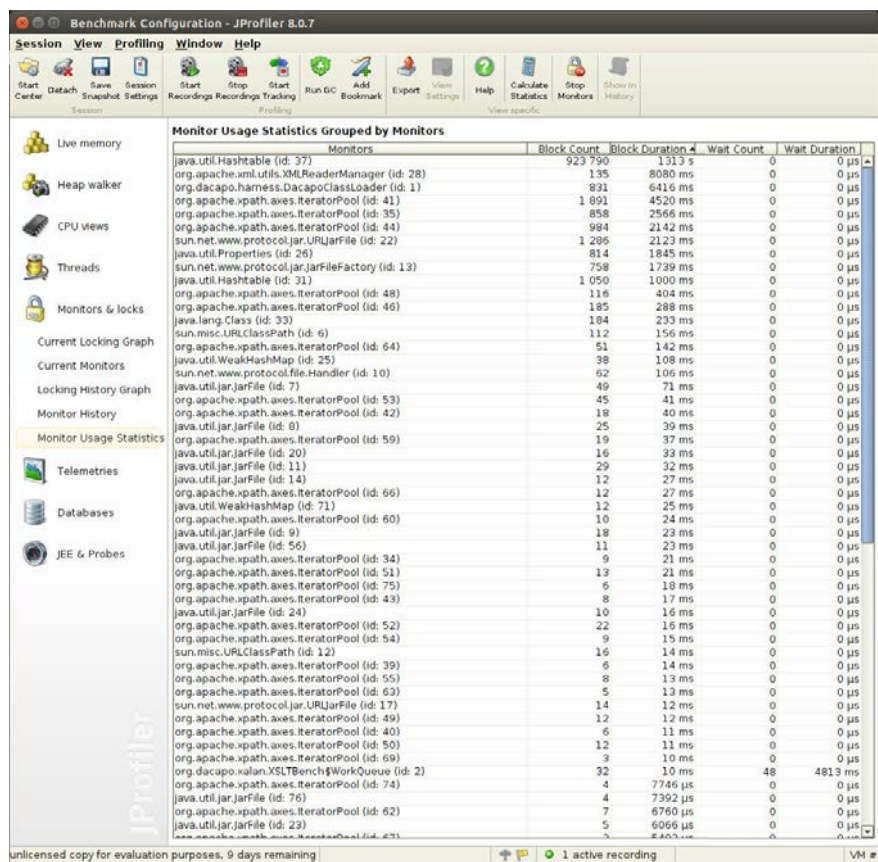
Figure 2.10: View of Monitor Usage Statistics, grouped by monitors, sorted by blocking duration.
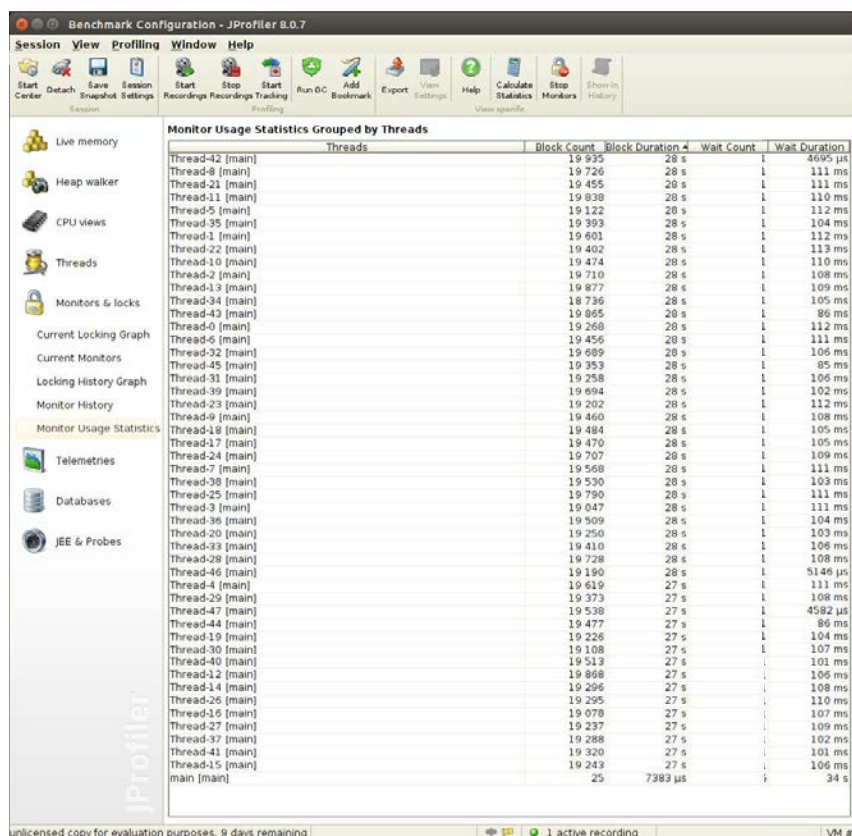
Figure 2.11: View of Monitor Usage Statistics, grouped by threads, sorted by blocking duration.
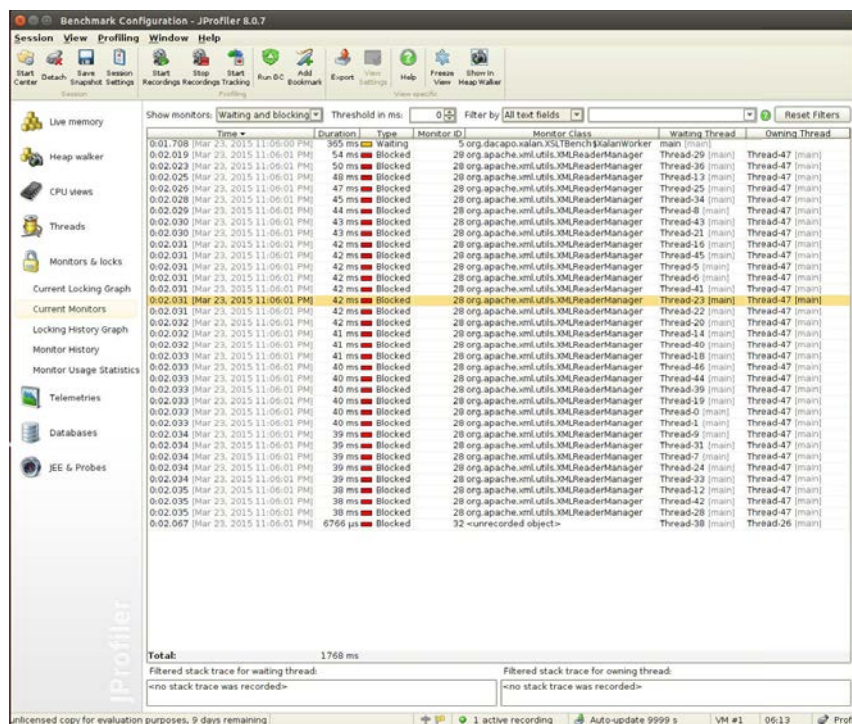
Figure 2.12: JProfiler view of current locked monitors.

2.2.3 Yourkit

Yourkit [97] is a general-purpose, commercial and closed-source profiler for Java developed by the Yourkit company since 2003. Yourkit and JProfiler shares many common characteristics. Yourkit features a graphical profiling interface with a dynamically-linked shared library for the JVM, provides a remote, local, and offline profiling mode, and can also be integrated into several IDEs. Yourkit supports CPU and thread profiling, memory profiling (including garbage collector), exception profiling, probes, deadlock detection and monitor usage profiling.

Using Yourkit for lock monitoring

Yourkit also profiles several components of the JVM by default at startup as JProfiler does. Likewise, it is advised to disable the profiling of components other than the locking subsystem since we are not interested in them and it could increase the overall application overhead. This is done by giving specific options on the command line of the JVM, which are as follows:

- monitors: this option starts monitor profiling directly at the startup of the Java application instead of starting it later from the graphical interface,

- disableexceptiontelemetry: this option specifies to not collect exception telemetry. The exception telemetry helps discovering performance issues and logic errors,

- disablestacktelemetry: this option specifies to not collect thread stack and status information shown in Thread view as well as in other telemetry views. This information allows the graphical interface to connect to the profiled application on demand and discover how the application behaved in the past,

- disablej2ee: this option specifies to disable J2EE profiling. This profiling inserts additional Java instructions into the Java bytecode,

- disabletracing: this option specifies to disable CPU tracing. This profiling also inserts additional Java instructions into the Java bytecode. Thus, only CPU sampling will be available,

- disablealloc: this option specifies to disable object allocation recording. This profiling also inserts additional Java instructions into the Java bytecode,

- builtinprobes=none: this option specifies to not register any of the built-in probes on startup (a predefined set of probes is available in Yourkit to help investigating typical problems and are ready to use out-of-the-box).

The command line to launch the application is therefore $ java −agentpath:~/yourkit−12.0.5/bin/ linux−x86−64/libyjpagent.so=monitors,disableexceptiontelemetry,disablestacktelemetry,disablej2ee,disabletracing ,disablealloc,builtinprobes=none java_program.

The Monitor Usage view shows statistics about locking and waiting operations acquired during the application profiling. It is organized in nested levels depending on how is sorted locking results, which is based on 3 criterion: blocker thread (thread that held the monitor

preventing the blocked thread from acquiring the lock), Waiting/Blocked thread (thread which called wait() and thread which failed to immediately acquire the lock), and monitor class (the Java class of the lock). Figure 2.13 presents results grouped by Blocker thread and then by Monitor class (a) or Waiting/Blocked thread (b), the Figure 2.14 presents results grouped by Monitor class and then by Blocker thread (a) or Waiting/Blocked thread (b), and the Figure 2.15 presents results grouped by Waiting/Blocked thread and then by Blocker thread (a) or Monitor class (b). Each view presents 2 types of information: which and for how long threads were calling wait(), and which and for how long threads were blocked on attempt to acquire a monitor held by another thread. A stack trace also presents the calling context that lead the thread to be waiting or to be blocked on a lock. It allows us to compute the Acquiring time of a lock divided by Elapsed time metric.
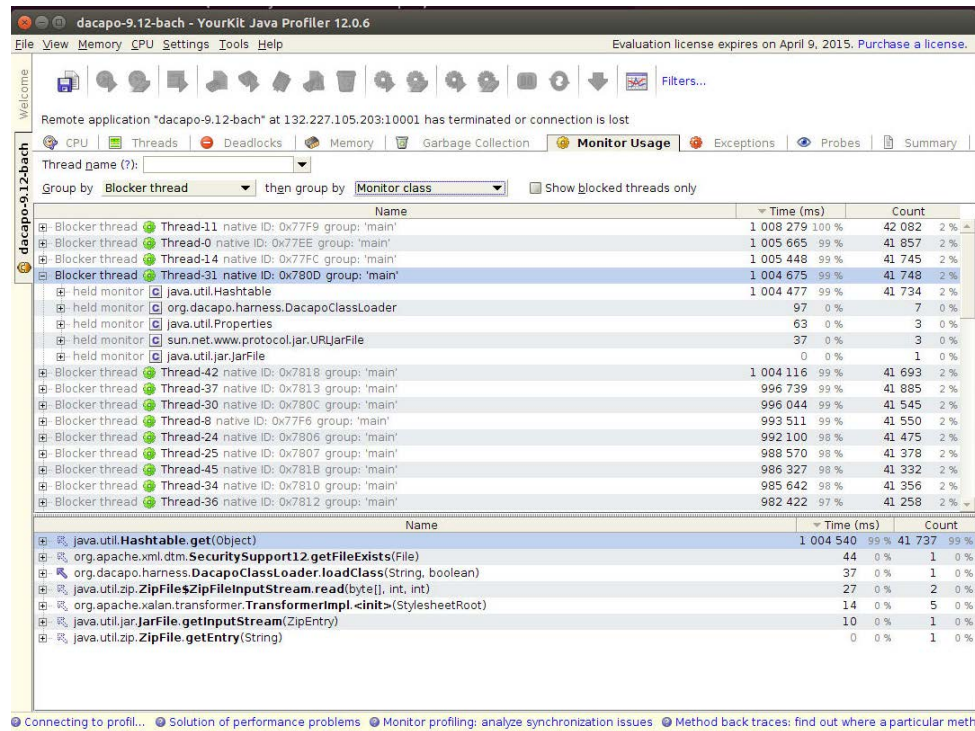
(a) Grouped by Blocker thread and by Monitor class.
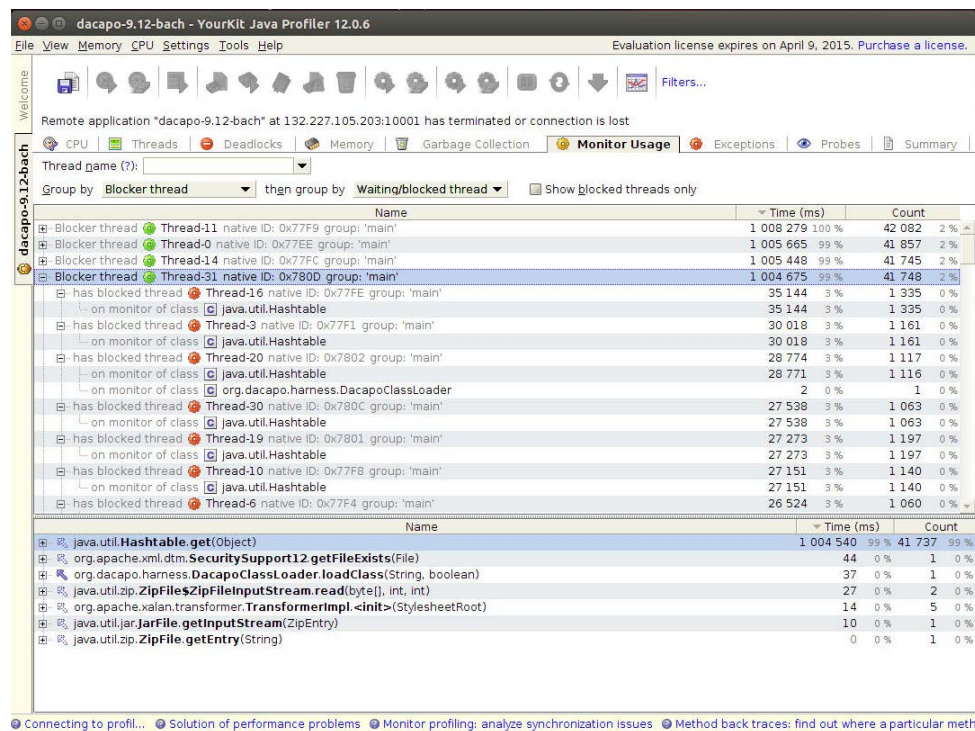
(b) Grouped by Blocker thread and by Waiting/Blocked thread.

Figure 2.13: Locks statistics for Yourkit, grouped by Blocker thread.
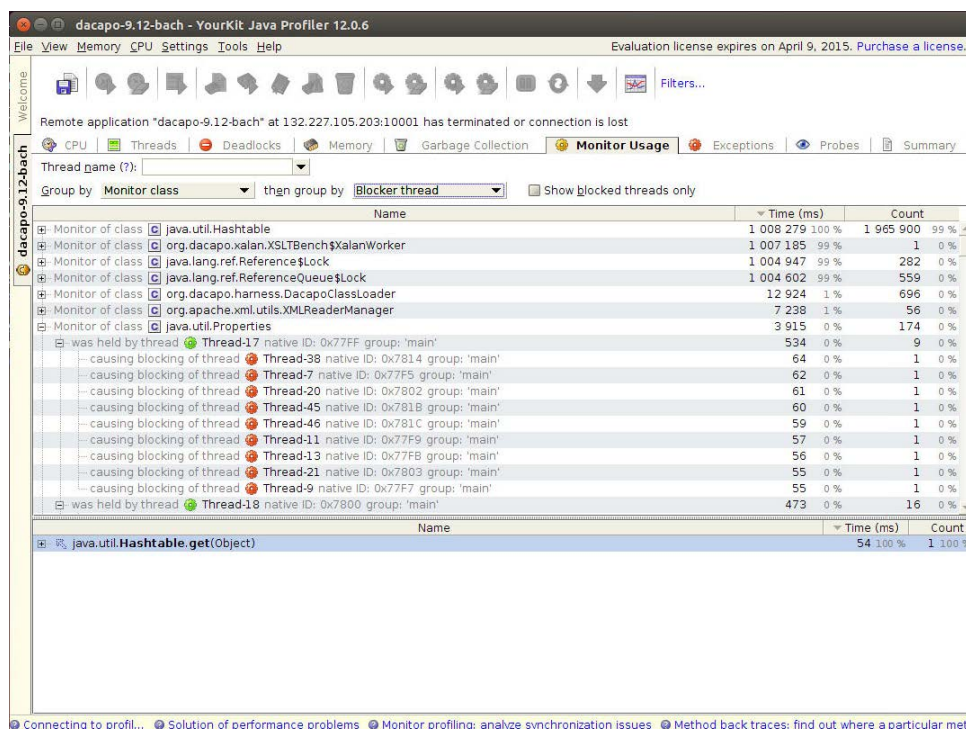
(a) Grouped by Monitor class and by Blocker thread.
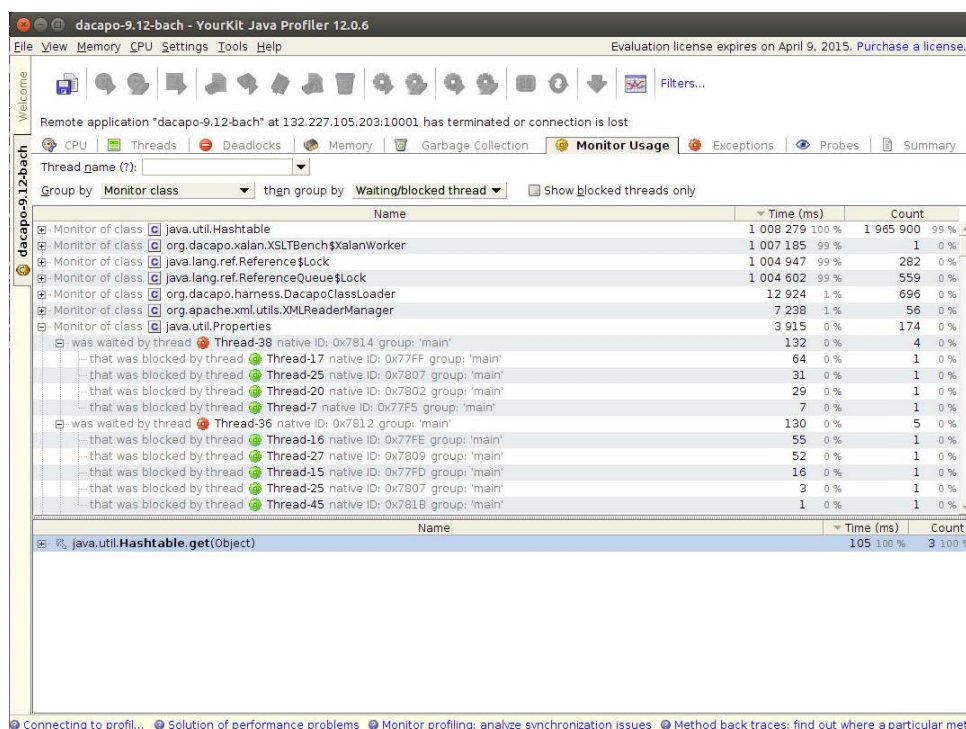
(b) Grouped by Monitor class and by Waiting/Blocked thread.

Figure 2.14: Locks statistics for Yourkit, grouped by Monitor class.
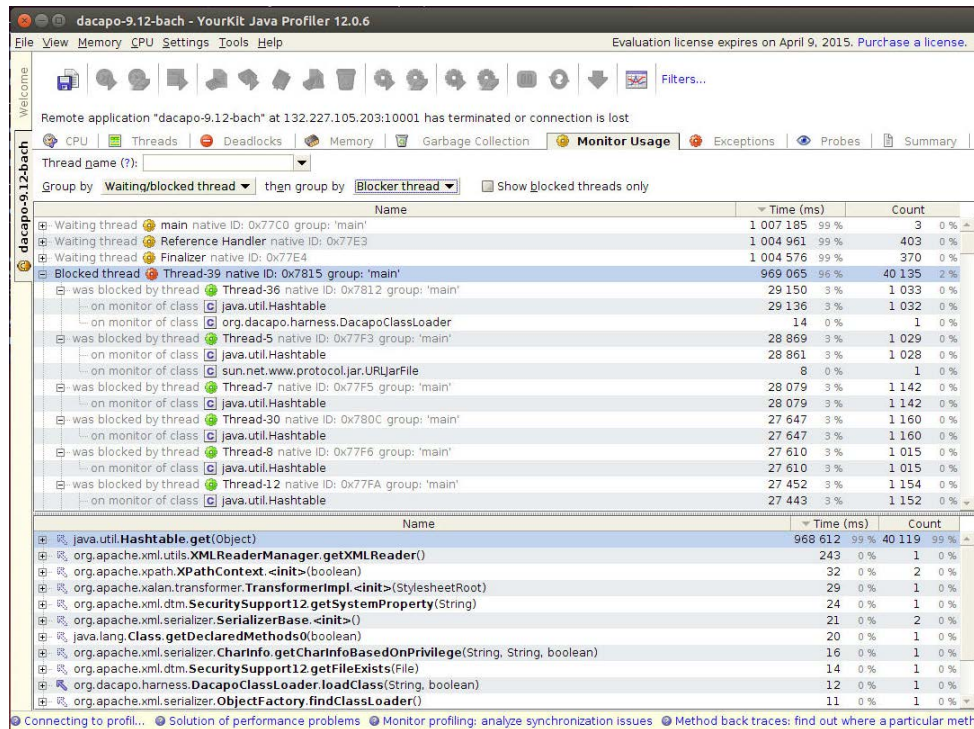
(a) Grouped by Waiting/Blocked thread and by Blocker thread.
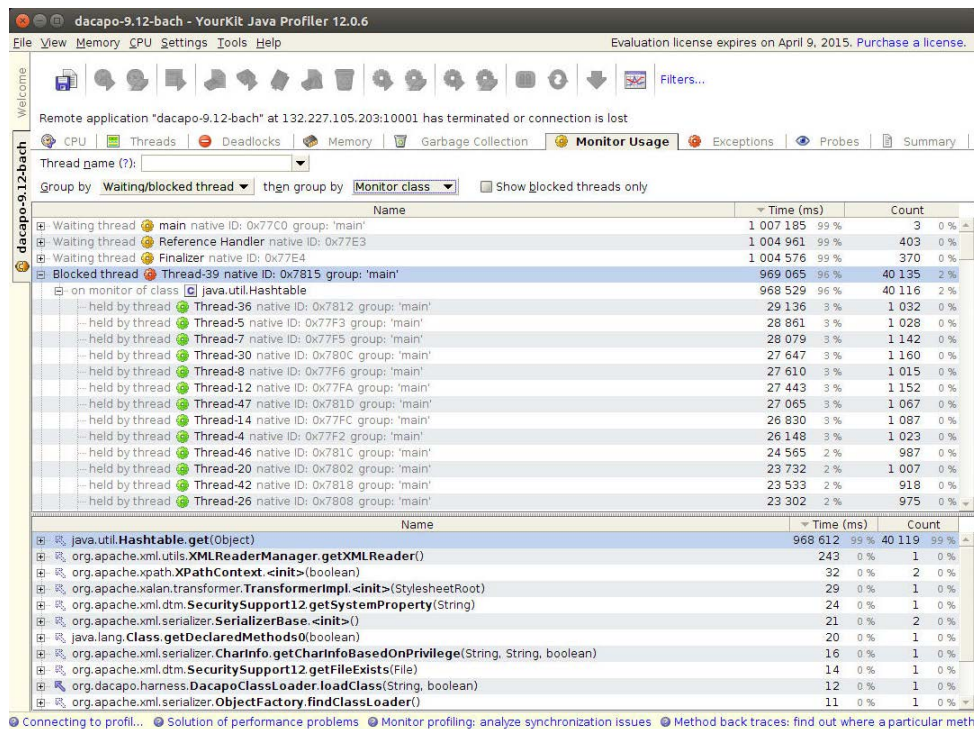
(b) Grouped by Waiting/Blocked thread and by Monitor class.

Figure 2.15: Locks statistics for Yourkit, grouped by Waiting/Blocked thread.

2.2.4 Health Center

Health Center [97] is a general-purpose, commercial and closed-source profiler for Java developed by IBM since 2003. It provides two components: a native dynamically-linked shared library shipped with the J9 JVM which gather profiling data and a graphical tool used to control the profiling process and analyze harvested data. This analysis tool can be either an Eclipse perspective or the IBM Support Assistant [44]. Health Center can offers distant profiling support with control of the profiler process at runtime via socket communication or in headless mode that logs data in a file later loaded in the graphical tool for analysis. Health Center works only with for the J9 JVM in order to achieve a low overhead due to very tight coupling with JVM internals and bypassing of JVMTI interfaces. Health Center supports CPU usage and thread profiling, garbage collector, class loading, I/O, method, and locking profiling.

Using Health Center for lock monitoring

Health Center is configured through the properties file located in jre/lib/healthcenter.properties. However, it does not provides the possibility to configure profiling features at a fine-grained level as in JProfiler or Yourkit where it is possible to disable profiling features that are not wanted. We choose to run the Health Center in headless mode, thus the command line is $ java − Xhealthcenter:level=headless java_program. The Eclipse addon is deployed into Eclipse by connecting to the IBM Eclipse addon repository and installing the *IBM Monitoring and Diagnostic Tools for Java - Health Center tool*. The file containing the profiled data (with an *.hcd extension) is then loaded into Eclipse by clicking on File and then Load data.

The Figure 2.16 presents the Locking perspective after loading the data. It brings 3 tabs together: Monitors, Monitors bar chart, and Analysis and Recommendations.

The Monitors tab presents statistics about all the locks found at runtime in the application. The description of the metrics is as follows:

- Gets: the total number of times the lock has been taken while it was inflated,

- Slow: the total number of non-recursive lock acquires for which the requesting thread had to wait for the lock because it was already owned by another thread,

- % miss: the percentage of the total Gets for which the thread trying to enter the lock on the synchronized code had to block until it could take the lock ($\% miss = (Slow \, / \, Gets) * 100$),

- Recursive: the total number of recursive acquires. A recursive acquire occurs when the requesting thread already owns the monitor,

- Average hold time: the average amount of time the lock was held or owned by a thread,

- % util: the amount of time the lock was held divided by the amount of time the output was taken over. The time the output was taken over refers to how long the application has been monitored for which is equivalent to the lifetime of the application ($100 * Average \, hold \, time * Slow \, / \, Application \, wall \, time$),
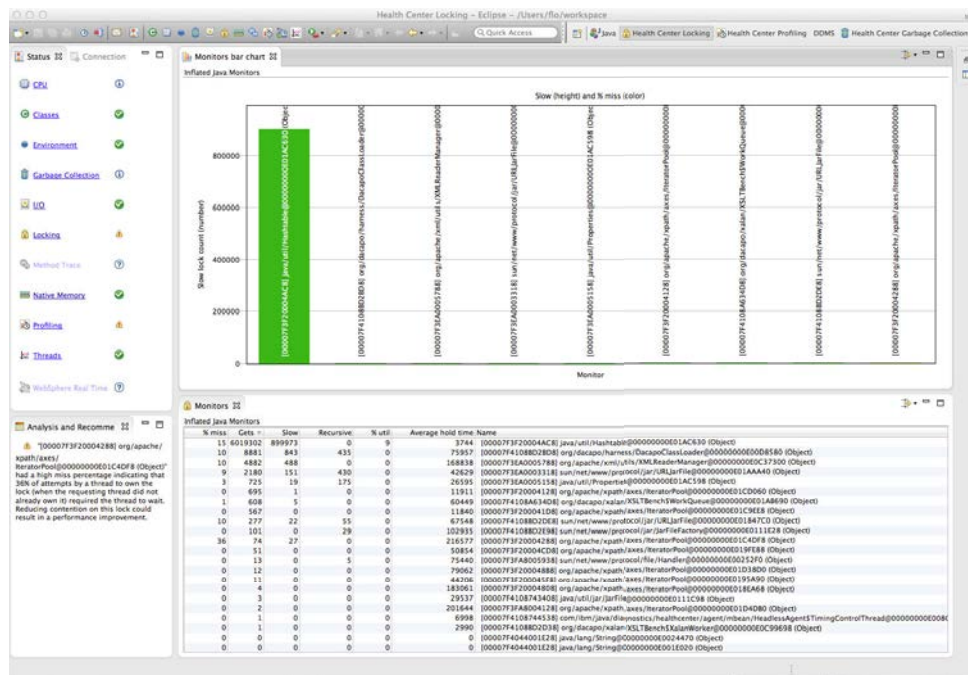
Figure 2.16: Locking view for Health Center.

- Name: The number in brackets is the object address, then follows the object class name and finally the object ID.

This tab allows us to compute the following metrics: Number of failed acquisitions divided by the Number of acquisitions (%*miss*), Number of failed acquisitions divided by the Elapsed time (*Slow* $*$ 100 / *Elapsed time*), and the total time spent in critical section of a lock divided by the Number of acquisitions (*Average hold time*).

The Analysis and Recommandations tab gives indications to the developers about potential candidates to blame for lock contention. A lock issue is detected when one of the metrics available in the Monitors tab exceeds a certain threshold. Then, this view reports the lock as problematic, tells the developer to try to optimize it and that doing so may lead to a performance improvement. The threshold values are unknown to the user and cannot be modified.

The Monitors bar chart shows an histogram of the most contended locks. The height of the bars represents the Slow lock count and is relative to all the columns in the graph. The color of each bar is based on the value of the % miss column in the table. The gradient moves from red (100%), through yellow (50%), to green (0%). A red bar indicates that the thread blocks every time that the monitor is requested and a green bar indicates a thread that never blocks.

2.2.5 Multicore Software Development Kit

Multicore Software Development Kit (MSDK) [69] is a closed-source Java profiler developed by the alphaWorks team [4] at IBM since 2006. It focuses on analyzing various concurrency related problems in applications which arise due to different types of memory and synchronization errors. MSDK consists of two parts: the msdk.sh command-line tool that executes programs

with different analysis engines and post-process profiling results, and an Eclipse perspective that can be used to launch programs and visualize the results. MSDK relies on JVMTI and also on a proprietary interface with the J9 JVM. For this reason, the lock analysis tool only works for the J9 JVM, otherwise MSDK is compatible with other JVMs.

Tools provided by MSDK are both static and dynamic analysis tools. Following is the set of tools included in MSDK:

- Race Detector: a debugging tool used to detect data races in the application code,

- Deadlock Detector: a debugging tool to find deadlocks in the application code,

- Static Concurrent Bugs Detector: a static analysis tool to find various concurrent bugs,

- Orange box Analysis: a tool to debug program crashes by providing the last few read/write values of variables by each thread,

- Lock Status Report: a tool to debug concurrent applications by providing information about threads holding locks and threads calling wait() in the code dynamically,

- Synchronization Coverage Analysis: a static analysis tool used to figure out whether the different synchronization primitives are doing something useful or whether it is redundant,

- MHP Analysis: this tool does a may-happen-in-parallel analysis and provides the statements that can execute in parallel.

Lastly, the Unified Lock Analysis tool profiles every types of locks found in the Java language. It provides an unified vision of locking by grouping synchronized blocks locks and locks from the java.util.concurrent package (analyzed by the JUCProfiler tool) into the same report. This allows the developer to have a global view of the application locking behavior by using a single tool.

Using MSDK for lock monitoring

The msdk.sh script is used to profile the application with the Unified Lock Analysis tool. The option −sync on profiles synchronized blocks and the option −juc on profiles java.util.concurrent locks. The command line to launch the application is thus $ ./bin/msdk.sh −sync on java_program.

MSDK generates 3 files when the application stops. For this run, the outcome is grouped into 3 files named 20150416112124.msdk, thor-20150416112124.dcagent.trace, and MSDKPost-Analyzer.log, the number in common being the date and time of the benchmark. Then, the post-processing of these 3 files is done with the same script. The option −post is used for that purpose and takes in parameter the file thor-20150416112124.dcagent.trace without the extension, thus giving the command $ bin/msdk.sh −post thor−20100929171822. The final result is finally written to the thor-20150416112124-Lock.txt file.

The Output 2.2 shows a typical output generated by MSDK shrunk for clarity. The first part (line 1 to 14) presents general information about the profiled application including MSDK version, system-wide information, the Java command line used to profile the application and JVM-specific information (this last information was not available on our system setup).

The second part (line 14 to 28) is the Java lock monitor report. This part presents lock statistic about synchronized locks in the Java program. The legend of this part is detailed in the third part (line 28 to 36). The MON-NAME information was not working on our system and is therefore not available in this report. It allows us to compute the following metrics: Number of failed acquisitions divided by the Elapsed time ($Slow * 100 / Elapsed\ time$), and the total time spent in critical section of a lock divided by the Number of acquisitions ($HELD\text{-}TIME\ / ENTER$).

The fourth part (line 36 to 41) contains information about java.util.concurrent locks found in the application. There is no profiling data in this part since JUC profiling was disabled.

The fifth part (line 41 to 101) is the Java monitor contention report. This list is ordered in 3 sublevels: by Syncpoint, by Monitor and by Threads. Each Syncpoint refers to a location in the source code and has an ID. This location is presented in details in the sixth part with the associated Java file name, method, line number, and Java class. Then, for each Syncpoint entry, there is a list of every Monitor that has been locked at this particular Syncpoint. Each Monitor has an ID that refers to an entry in the seventh part where the class of the Monitor can be found. Finally, for each Monitor entry, there is a list of every Thread that has locked this particular Monitor at this particular Syncpoint. Each Thread is presented with its name, its ID, the number of lock acquisitions (CT-TIMES), the total duration of locking (CT-DURATION), the number of call to wait() (WT-TIMES), and the total duration spent in calls to wait() (WT-DURATION).

The sixth part (line 101 to 108) is about Contention Information. It shows every Syncpoint in the Java monitor contention report with its associated Java file name, method, line number, and Java class. The seventh and last part (line 108 to 115) is about Java monitor information. It lists every Monitor in the Java monitor contention report and its associated Java class.

Output 2.2: Profiling output of MSDK

```
 1 ================================================================================
 2 Multicore Software Development Tookit Version_2.1
 3 Reader Version: v2.2.0.   Build Time: 20101207−1609
 4
 5 PROPERTIES:
 6 Arch:     OS:      Version: 3.9.0−replication+    Host: amd48c−systeme    TimeStamp:
     ↪ 93597694055359
 7
 8 JavaCommandLine:
 9 /home/florian/java−rcl/profilers/j9/ibm−java−x86_64−71//bin/java −agentpath:/home/florian
     ↪ /MSDK/lib/libThorAgent.so=traceSyncLock=on,enableJLM=on −Xbootclasspath/p:/home/
     ↪ florian/MSDK/lib/BCIRuntime.jar:/home/florian/MSDK/lib/PreInstrument.jar −
     ↪ javaagent:/home/florian/MSDK/lib/BCIAgent.jar=callStackDepth=10,
     ↪ allocationStackDepth=10,traceJUC=off,msdk.idp=com.ibm.msdk.bciagent.
     ↪ JUCInstrumentDecisionProvider,msdk.lib=/home/florian/MSDK/lib −cp .: −jar /home/
     ↪ florian/java−rcl/benchmarks/dacapo−9.12−bach.jar −s large xalan
10
11 JavaFullVersion:
12
13
14 ================================================================================
15 Java Lock Monitor Report
16
17       SPIN2        SLOW       ENTER      YIELDS          REC        HELD−TIME  MON−NAME
```

18	8191008	1402638	2972991	4125206	0	305489225625
19	181209	112	1678	5832	556	103214622
20	166424	639	6810	7102	416	714123416
21	164625	5164	51642	16439	0	1183548818
22	140937	13	2076	4016	0	48569814
23	117452	68	312	3674	61	125556875
24	19217	0	152	558	0	8851772
25	17900	15	19	526	0	5827679
26	16353	0	40	483	16	4717132
27	10356	1	33	303	6	3691277

```
29  LEGEND:
30          SPIN2 : Number spins before this monitor became inflated
31            REC : Number of times this monitor is entered by the same thread
32           SLOW : Number of times this monitor is entered via the slow path
33          ENTER : Number of times this monitor is entered
34         YIELDS : Number of calls to yield() before this monitor became inflated
35      HELD-TIME : The total time this monitor is in held state
```

```
38  j.u.c Lock Profiler Report

40  No lock contention found in using Juc

42  Java Monitor Contention Report
```

	SYNCPOINT (ID)	MONITOR (ID)	THREAD (ID)	CT-TIMES	CT-DURATION	WF-TIMES	WF-DURATION
44	SYNCPOINT (ID)	MONITOR (ID)	THREAD (ID)	CT-TIMES	CT-DURATION	WF-TIMES	WF-DURATION
45	SyncPoint (435)			1453355	25230828712384	0	0
46		Monitor (48)		1453347	25230803069423	0	0
47			Thread-19 (77)	30642	525931329282	0	0
48			Thread-28 (86)	30635	525729569705	0	0
49			Thread-18 (76)	30595	524691337459	0	0
50			Thread-37 (95)	30587	524694533587	0	0
51			Thread-33 (91)	30579	526604213543	0	0
52			Thread-48 (106)	30544	527340799238	0	0
53			Thread-16 (74)	30514	526139730529	0	0
54			Thread-14 (72)	30495	527390108230	0	0
55			Thread-45 (103)	30484	524932002770	0	0
56			Thread-42 (100)	30450	526878327517	0	0
57			Thread-10 (68)	30446	525114979908	0	0
58			Thread-51 (109)	30440	524078754034	0	0
59			Thread-53 (111)	30431	526478071738	0	0
60			Thread-36 (94)	30392	526235761437	0	0
61			Thread-34 (92)	30368	525587970274	0	0
62			Thread-11 (69)	30364	525966784145	0	0
63			Thread-46 (104)	30334	525159893999	0	0
64			Thread-47 (105)	30332	525089281170	0	0
65			Thread-26 (84)	30328	526083662252	0	0
66			Thread-12 (70)	30328	526301184170	0	0
67			Thread-29 (87)	30318	525625704796	0	0
68			Thread-13 (71)	30317	525637160184	0	0
69			Thread-23 (81)	30309	525196317168	0	0
70			Thread-44 (102)	30288	526928679920	0	0
71			Thread-31 (89)	30286	525937926542	0	0
72			Thread-41 (99)	30269	525801795621	0	0
73			Thread-21 (79)	30267	524487216089	0	0
74			Thread-7 (65)	30264	525383312006	0	0
75			Thread-32 (90)	30262	524486703777	0	0
76			Thread-8 (66)	30230	525180332559	0	0
77			Thread-20 (78)	30229	523930802380	0	0
78			Thread-49 (107)	30224	525890241848	0	0

```
 79                                Thread−17 (75)      30208   525384939266        0          0
 80                                Thread−27 (85)      30196   525731840652        0          0
 81                                Thread−52 (110)     30172   524512067912        0          0
 82                                Thread−25 (83)      30165   525559954364        0          0
 83                                Thread−38 (96)      30165   525367905346        0          0
 84                                Thread−39 (97)      30159   526897004611        0          0
 85                                Thread−50 (108)     30100   526165670705        0          0
 86                                Thread−22 (80)      30098   526540272937        0          0
 87                                Thread−40 (98)      30057   525021061457        0          0
 88                                Thread−30 (88)      30045   525369112680        0          0
 89                                Thread−54 (112)     29998   525136003820        0          0
 90                                Thread−24 (82)      29958   524177721905        0          0
 91                                Thread−15 (73)      29937   525843715814        0          0
 92                                 Thread−9 (67)      29906   525980352991        0          0
 93                                Thread−43 (101)     29869   525694158448        0          0
 94                                Thread−35 (93)      29763   526506798638        0          0
 95              Monitor (46)                              8       25642961        0          0
 96                                Thread−34 (92)          3         862163        0          0
 97                                Thread−17 (75)          2       17708275        0          0
 98                                Thread−31 (89)          1        3303808        0          0
 99                                Thread−21 (79)          1        3757930        0          0
100                                Thread−14 (72)          1          10785        0          0
101 ═══════════════════════════════════════════════════════════════════════════════════════
102 Contention Information:
103
104           ID                JAVA FILE        METHOD  LINE      CLASS
105           ...                   ...            ...    ...      ...
106     SyncPoint435        Hashtable.java         get   479      Ljava/util/Hashtable;
107           ...                   ...            ...    ...      ...
108 ═══════════════════════════════════════════════════════════════════════════════════════
109 Java Monitor Information:
110
111           ID      MONITOR CLASS
112           ...      ...
113     Monitor48      Ljava/util/Hashtable;
114           ...      ...
115 ═══════════════════════════════════════════════════════════════════════════════════════
```

2.2.6 Java Lock Monitor

Java Lock Monitor (JLM) [67] is a closed-source Java profiler included in the Performance Inspector toolkit [67], a suite of performance analysis tools for Java and C++ applications, and works for the J9 JVM exclusively. It was developed by IBM from 2003 until July 2010, the date of the last release, and is no longer maintained. JLM relies on JPROF, a profiling agent shipped as a dynamic library with the J9 JVM that interfaces with events from either JVMTI or JVMPI, and with a command line tool to control the profiling process.

JLM focuses on execution and data profiling in different ways. Execution profiling exists in 3 styles: time profiling, callflow profiling, and callstack sampling. On the other hand, data profiling is done either by heap dump analysis, after which an offline analysis reports information about the heap, or by simply tracking object allocations and deallocations occuring in every method.

Using Java Lock Monitor for lock monitoring

The JVM has to be launched with the JPROF profiling agent enabled with one of the 2 following commands: *#java −agentlib:jprof java_program* or *#java −Xrunjprof java_program*. Then the rtdriver program connects to the JVM to control the profiling process and start collecting profiling data on demand. It is possible to start collecting locking data by sending the jlmstart command, to stop collecting locking data by sending the jlmstop command, and to dump locking data to a file by sending the jlmdump command. Data are dumped to files named log-jlm.#_pppp where # is a sequence number starting with 1, and pppp is the PID of the Java process.

Output 2.3: Profiling output of Java Lock Monitor

```
1    JLM_Interval_Time  28503135251
2
3   System (Registered) Monitors
4   %MISS    GETS NONREC   SLOW    REC TIER2 TIER3 %UTIL AVER-HTM  MON-NAME
5      11      91     91     10      0     0     0     1   4550728  JITC Global_Compile lock
6       0    2466   2217      0    249     0     0     0      1780  Thread queue lock
7       0     752    751      0      1     0     0     0     11160  Binclass lock
8       0     701    695      0      6     0     0     0     71449  JITC CHA lock
9       0     286    286      0      0     0     0     0    408679  Classloader lock
10      0     131    131      0      0     0     0     0     26877  Heap lock
11      0      61     61      0      0     0     0     0      2188  Sleep lock
12      0      51     50      0      1     0     0     0       718  Monitor Cache lock
13      0       7      7      0      0     0     0     0       608  JNI Global Reference Lock
14      0       5      5      0      0     0     0     0       780  Monitor Registry lock
15      0       0      0      0      0     0     0     0         0  Heap Promotion lock
16      0       0      0      0      0     0     0     0         0  Evacuation Region lock
17      0       0      0      0      0     0     0     0         0  Method trace lock
18      0       0      0      0      0     0     0     0         0  JNI Pinning lock
19
20  Java (Inflated) Monitors
21
22  %MISS    GETS NONREC   SLOW    REC TIER2 TIER3  %UTIL AVER-HTM   MON-NAME
23     33       3      3      1      0     0     0      0     8155   java.lang.Class@7E8EF8/7
       ↪ E8F00
24     33       3      3      1      0     0     0      0     8441   java.lang.Class@7E8838/7
       ↪ E8840
25      0 3314714 3314714    809      0     0     0      3      278   testobject@104D3150/104
       ↪ D3158
26      0 3580384 3580384    792      0     0     0      4      281   testobject@104D3160/104
       ↪ D3168
27      0       1      1      0      0     0     0      0      735   java.lang.ref.
       ↪ ReferenceQueue$Lock@101BDE50/101BDE58
28      0       1      1      0      0     0     0      0      833   java.lang.ref.
       ↪ Reference$Lock@101BE118/101BE120
```

The Output 2.3 shows a typical output generated by JLM when the jlmdump command is actually entered. The JLM_Interval_Time variable at the top represents the time interval between the jlmstart and the jlmdump. This time is expressed generally in cycles but can vary depending on the hardware platform. The output is then divided in 2 distinct parts with the system monitors first and the Java monitors then.

The signification of each field in the output is as follows:

- %MISS: the percentage of the total GETS (acquires) where the requesting thread was blocked waiting on the monitor ($\%MISS = (SLOW \ / \ NONREC) * 100$),

- GETS: the total number of successful acquires ($GETS = FAST + SLOW + REC$),

- NONREC: the total number of non-recursive acquires. This number includes SLOW gets,

- SLOW: the total number of non-recursive acquires which caused the requesting thread to block, waiting for the monitor to become unlocked. This number is included in NONREC,

- REC: the total number of recursive acquires. A recursive acquire is one where the requesting thread already owns the monitor,

- TIER2: the total number of inner spin loop iterations on platforms that support backoff-spinning,

- TIER3: the total number of outer thread yield loop iterations on platforms that support backoff-spinning,

- %UTIL: the monitor hold time divided by JLM_Interval_Time,

- AVER-HTM: the average amount of time the monitor was held. Recursive acquires are not included because the monitor is already owned when acquired recursively ($AVER-HTM = Total\ hold\ time\ /\ NONREC$),

- MON-NAME: the system monitor name for system monitors or the Java class and the memory address of the object associated with the monitor for Java monitors.

Therefore, as with Health Center, this output allows us to compute the following metrics: Number of failed acquisitions divided by the Number of acquisitions (%$MISS$), Number of failed acquisitions divided by the Elapsed time ($SLOW * 100\ /\ Elapsed\ time$), and the total time spent in critical section of a lock divided by the Number of acquisitions ($Average\ hold\ time$).

There are very few differences between JLM and HealthCenter in terms of the data presented, as they both gather the data in the same way. HealthCenter however has the added level of helping to interpret the data thanks to the graphical interface and to alert about potential locks to blame for lock contention and causing a performance bottleneck.

2.2.7   Java Lock Analyzer

Java Lock Analyzer (JLA) [49] is a lock profiler which provides a real-time and dynamic lock monitoring on live Java applications for the J9 JVM developed by the alphaWorks team [4] at IBM from 2003. The project is not supported by IBM anymore since around 2007 and JLA is not available for download on the IBM website as well.

JLA consists of 2 parts called JLAagent and JLAGui. The JLAagent component is loaded dynamically by the JVM on startup and gather the lock information on the running application. The JLAGui is a graphical interface of the lock analysis statistics. JLA can profile applications locally or remotely. It is also specific to the J9 JVM and is not compatible with any other JVM. It was not possible to run JLA on the latest J9 JVM since the development of JLA has been stopped several years ago, thus making them not compatible together anymore. JLA has since been superseded by HealthCenter and MSDK.

Using JLA for lock monitoring

The JVM must load the JLAagent at startup for profiling, this is done with the command *#java −*
Dcom.sun.management.config.file=JLAtiagent.properties −agentlib:JLAtiagent −cp .;JLAagent.jar java_program
. The JLAtiagent.properties property file allows the developer to specify options for the remote
JVM connection like using SSH to communicate or the remote port of connection.

The first view of the graphical interface of JLA presents information similar to the Monitors
bar chart of Health Center, namely an histogram of the most contended locks. The height of
each column is calculated on the value of the SLOW lock count and is relative to all the columns
in the graph. The colour of each bar is based on the %MISS value with a gradient going from
red (100%), through yellow(50%) and finally onto green (0%). A red bar indicates that the
thread blocks every time the monitor is requested whereas a green bar indicated the thread
never blocks.

The second view of the interface shows the same information as JLM presented earlier (GETS,
NONREC, SLOW, NONREC, REC, TIER2, TIER3, %UTIL, AVER-HTM, and MONITOR
NAME). Therefore, as with JLM and Health Center, it is possible to compute the Number
of failed acquisitions divided by the Number of acquisitions metric (*%miss*), the Number of
failed acquisitions divided by the Elapsed time metric (*Slow* ∗ 100 / *Elapsed time*), and
the total time spent in critical section of a lock divided by the Number of acquisitions metric
(*Average hold time*).

2.3   Related work

This section presents state-of-the-art of profiling in the context of concurrent applications run-
ning on multicore hardware. The first part describes lock profilers found in the literature other
than the 7 locks profilers presented previously. The second part presents profilers tailored to
solve problems typically found in parallel applications on multicore architectures other than lock
contention issues.

2.3.1   Lock profilers in the literature

Inoue et al. [45] have proposed a sampling-based profiler relying on Hardware Performance
Monitor that collect object creation and lock activity profiles. The profiler uses a dedicated
instruction, called *ProbeNOP*, inserted in the Java code by the JIT compiler at runtime with the
Oprofile driver, an interface to set the HPM-related special purpose registers. The profiler will
sample this specific instruction on a regular basis, depending on the sampling rate. A sample is
collected by a sampling-handler that records information about the address of the Java object
used for locking and the calling method. This approach achieves a low overhead of less than 2.2
% but since it uses the same metric as Health Center, it suffers from the same limitations.

For C applications, Mutrace [70] and the profiler used in RCL [64] profiles locks from the
POSIX API. Mutrace reports metrics similar to what is found in Health Center. The RCL
profiler uses the total time to acquire the lock (blocking time included), execute the critical
section itself, and release the lock as a metric to know if a lock can benefit from RCL. These
profilers also have the same limitations of the metrics presented in Section 3.1.

WAIT [5] is a sampling-based tool that diagnoses various performance issues in running server-class applications in order to understand the cause of thread idleness. 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 we found in Cassandra or Xalan.

Xian et al. [95] 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 [14] is a tool that targets spinlock profiling for the Linux kernel. Like, e.g., Java Lock Monitor [67], Lockmeter reports the time spent in the critical section protected by a spinlock divided by the elapsed time. As shown in Section 3.1, this metric does not report a useful value on some synchronization patterns.

HPCToolkit [88] is a sampling-based profiler designed for high performance computing. The authors define a new metric called *Blame shifting*: this metric attributes idleness incurred by threads waiting to acquire a lock directly to the calling context of the lock holder. A common point in this work and Free Lunch is to require an auxiliary data structure to store profiling data. However, unlike in Java where a data structure is automatically created when lock contention is encountered, there is no dual-representation locks [8] (flat and inflated) in C applications. HPCToolkit therefore defines a protocol to associate and to maintain a data structure in order to monitor a lock. Once monitored, a lock remains in inflated mode, which could be a performance issue in Java where the lock goes back to flat mode if no contention is experienced after some time. As a consequence of the metric design, HPCToolkit needs an atomic addition at every sample (to compute the delay for acquiring the lock by each thread) and a Compare-And-Swap for lock release (for blaming the thread holding the lock). Free Lunch does not require any additional synchronisation or memory barrier to the baseline lock implementation. Nevertheless, the overhead of HPCToolkit manages to remain below 5 %. This metric is complementary to Free Lunch in the sense that HPCToolkit attributes lock contention to threads whereas Free Lunch measures lock-related CSP.

`Java.util.concurrent` is a Java API that provides lock-free data structures. JUCProfiler (which is part of MSDK [69]) and JProfiler [52] are able to profile such libraries. Free Lunch does not currently provide this type of profiling.

Finally, HaLock [43] 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.

2.3.2   Other profilers for parallel applications

Capacity planning [68] 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 authors 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 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, it could be used as a building block for capacity planning.

COZ [20] is a profiler that indicates exactly where programmers should focus their optimization efforts. It relies on casual profiling, a technique that *virtually speedup* fragments of code by slowing down all other code running concurrently. Developers just need to insert progress points in the source code where some useful unit of work is completed. COZ runs several performance experiments during a program's execution where selected fragments of code are virtually speedup. The profiler returns the whole application speedup due to the impact of optimizing a particular code fragment by a specific percentage. Free Lunch and COZ approaches are complementary: a developer can use Free Lunch to detect a lock contention problem and then use COZ to evaluate the application speedup he can expect from improving the incriminated lock.

Bottle Graphs [32] 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. [56] analyze communication patterns of shared Java objects and define new kind of concurrency metrics that they apply to the DaCapo benchmark suite [12]. 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 [28] 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 3.3.1, we have observed that the drift between the CPUs is not significant.

Memprof [60] and Carrefour [21] focus on optimizing memory accesses on Non-Uniform Memory Architecture (NUMA) hardware. Congestion on memory controllers and interconnects on

NUMA systems is a source of application slowdown. Carrefour implements in-kernel memory management algorithms that dynamically profile applications by gathering various metrics about the application memory behavior. Based on these data, it applies several memory placement algorithms to lower memory traffic congestion. Memprof identifies typical memory access patterns harmful for NUMA architecture by building temporal flows of memory access between threads and objects. It pinpoints the offending fragment of code and helps developers to easily modify the source code in order to reduce remote data accesses. These works focus on memory placement issues for NUMA hardware whereas Free Lunch finds lock contention issues. All of these tools are suitable to belong to a standard toolbox to find performance problems on NUMA multicore hardware.

DProf [78] and Sheriff [63] aims to find cache related performance issues. DProf associates the cache miss costs and reasons to data types and presents its results with several comprehensive views to developers. It is able to differentiate caches misses causes and allow developers to apply the proper solution to solve these problems. Sheriff focuses on detecting false sharing between threads by transforming them into OS processes and leveraging the OS memory protection mechanism with the Sheriff framework. A first tool, Sheriff-Detect, reports with no false positives instances of false sharing by comparing updates within the same cache lines by different threads. A second tool, Sheriff-Protect, eliminates false sharing by delaying concurrent updates to a conflicting cache line to the next synchronization, even with the absence of the source code. These tools are appropriate to detect issues about cache induced performance issues in multithreaded applications and like Memprof, Carrefour, and Free Lunch, are suitable to belong to a standard toolbox to find performance problems on NUMA multicore hardware.

There is many profilers dedicated to High Performance Computing (HPC). Most performance analysis approaches consist in tracing the application behavior (using tools like Tau [85], Vampir-Trace [71], EZTrace [34, 92, 93], or Extrae [35]): calls to a predefined set of functions (typically, MPI or OpenMP primitives) are recorded in a trace file. The resulting execution traces can be analyzed *post-mortem* in order to find the application bottlenecks [9, 18, 46, 73, 79, 84, 90]. This can be done manually by the application developper using a trace visualization software (for instance Vampir [73], ViTE [18] or Intel Trace Analyzer and Collector [46]). A basic approach towards automatic analysis of execution traces have also been implemented recently in multiple tools: for instance, Scalasca [90] or Periscope [9] search through a collection of typical inefficient patterns of events related to MPI communications or through a collection of typical OpenMP synchronization problems. Thus, while many performance analysis tools exist for HPC applications, most of them are MPI-centric and the employed techniques can only be applied to HPC applications. The automatic detection of performance bottlenecks with such tools is based on databases of classical HPC performance problems, such as MPI synchronization, OpenMP concurrency, cache usage, etc. These profilers are dedicated to profiling HPC applications while Free Lunch is dedicated to profiling lock issues in Java applications.

Other approaches investigated recently consist in analyzing system logs [2, 15, 17, 72, 81, 100]. This allows developers to pinpoint issues that involve multiple layers of the software stack with a very coarse granularity. For example, these approaches can help detecting that there is a bottleneck within the network infrastructure, or within a given machine or in a macro-scale software component. Free Lunch works at a fine granularity in order to precisely pinpoint the

offending code fragment.

Chapter 3

The Free Lunch profiler

This chapter presents a survey of lock metrics of lock profilers presented in the previous chapter, the Critical Section Pressure metric, and the Free Lunch profiler. Section 3.1 discusses the limitations of the metrics of the seven state-of-the-art Java lock profilers of which we are aware in the context of Java server profiling. This study influences directly the idea of the CSP metric and consequently, the design and the implementation of Free Lunch. Section 3.2 presents the definition of the Critical Section Pressure metric and how it assesses the impact of locks on thread progress, and the way the computation frequency of the CSP was defined. Section 3.3 describes the implementation of the Free Lunch profiler in the Hotspot JVM. In particular, it focuses on the times composing the CSP, how to gather them, and how to minimize the overhead when collecting and storing them.

3.1 Lock contention metrics

In this section, we study the metrics used by the seven state-of-the-art profilers presented in Section 2.2. 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.

3.1.1 Synchronization scenarios

The *generalized ping-pong scenario*, presented in Figure 3.1, 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.
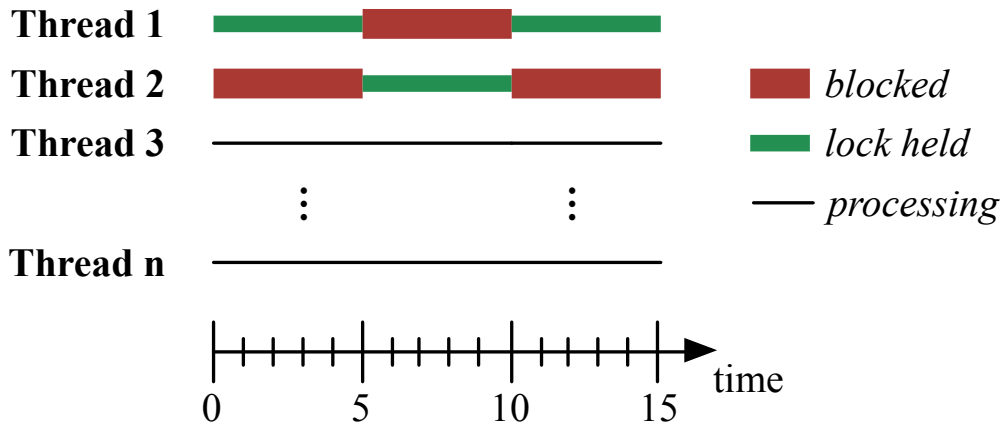

Figure 3.1: A generalized ping-pong scenario.

In the *fork-join scenario* shown in Figure 3.2, 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.
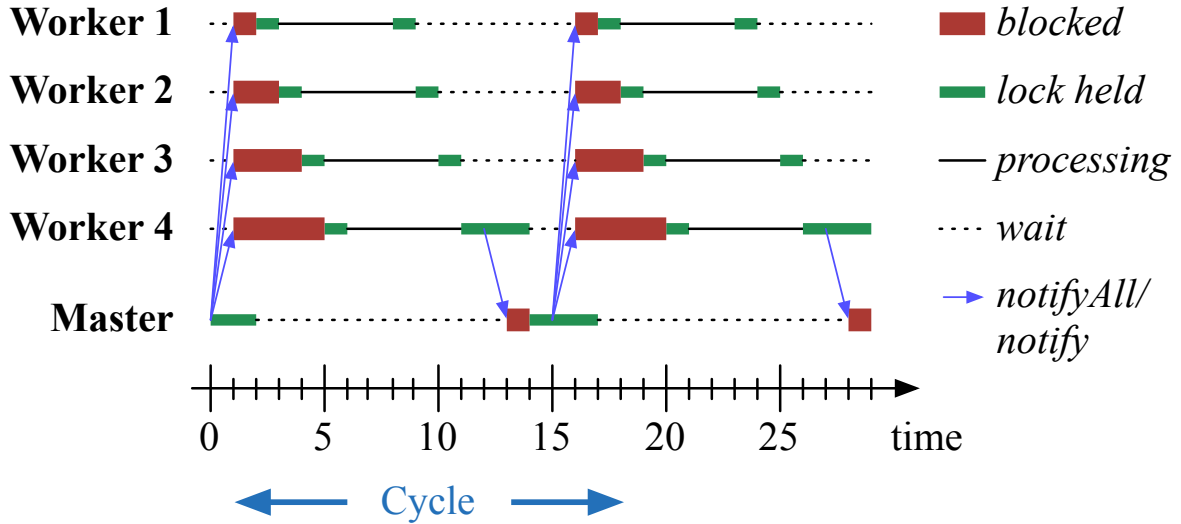

Figure 3.2: A fork-join scenario.

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.

3.1.2 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 3.1 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 4, 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 3.2), 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 3.1). 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 3.1), regardless of the number of threads, the average time spent in critical sections (the duration of the green thick 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

Table 3.1: 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

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.

3.2 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.

3.2.1 Critical Section Pressure 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 3.1). 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 3.1), 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 acquisition, reflecting the fact that the lock does not prevent application progress. For the fork-join scenario (Figure 3.2), 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.

3.2.2 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.
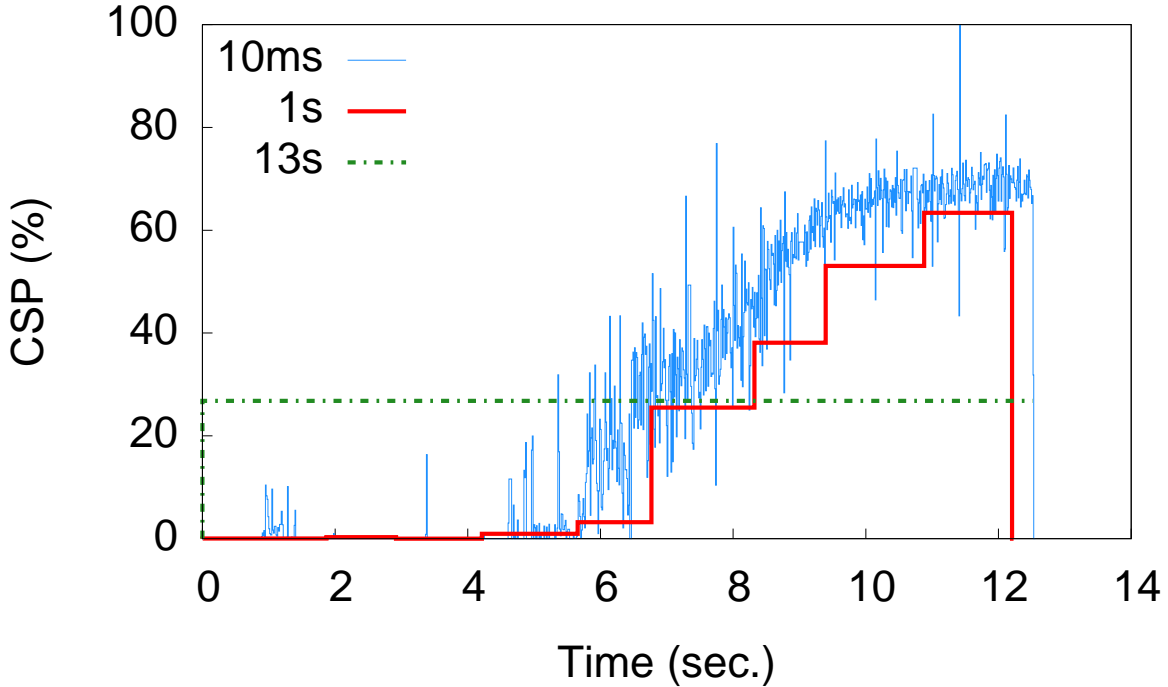

Figure 3.3: CSP depending on the minimal measurement interval for the Xalan application.

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 3.3 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.

3.2.3 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 [5] and our experience in analyzing the Java programs described in Section 4.3.2 shows that only a single call stack is generally sufficient to understand why a lock impedes thread progress.

3.2.4   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 [96] or lock-free algorithms [62].

Furthermore, Free Lunch has no application-specific information about the role of the individual threads, and thus assumes that all threads are equally important to the notion of progress. For example, in the generalized ping-pong scenario, it may be that the two ping pong threads control output to the user, and the remaining threads perform computations whose results will be ultimately discarded, if the ping pong threads cannot output them sufficiently quickly. A low CSP for this scenario would not reflect the user experience.

3.3   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.

3.3.1   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 3.4). 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 4.1.2) is the use of a map that associates each Java object to its profiling data. As was for example proposed in RaceTrack [98], 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.
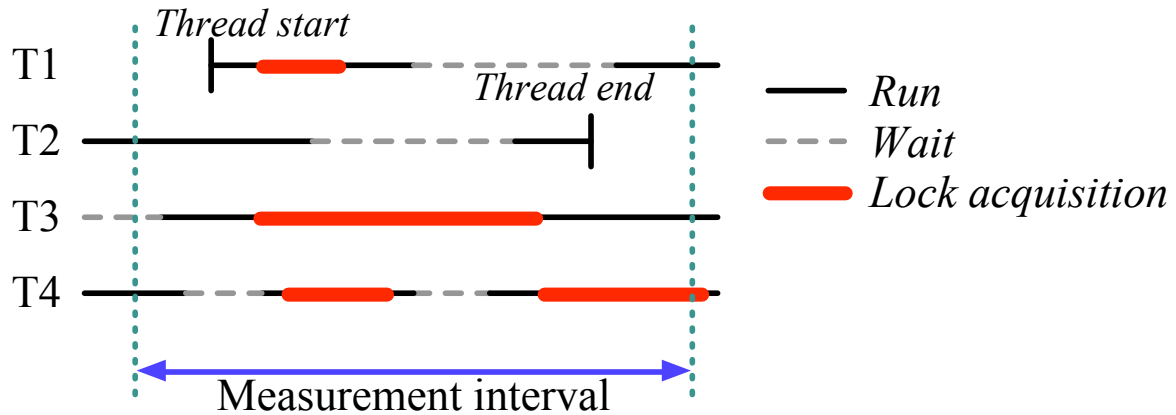
Figure 3.4: Time periods relevant to the CSP computation.

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 [31], introduce an additional costly instruction to prevent reordering. Fortunately, a Java lock acquisition triggers a full memory barrier [65], 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 3.2.1, 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.

3.3.2   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.[1] 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.

3.3.3   Limitations of our implementation

Storing profiling data inside the monitor data structure in Hotspot 7 is not completely reliable, because deflation can 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 [51]. Such a collector can move the object to a different address while the monitor is deflated. In this case, Free Lunch

---

[1]We have used this option for the experiment presented in Figure 3.3.

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.

Chapter 4

Performance evaluation

We now evaluate the performance of Free Lunch as compared to the existing profilers for Open-JDK: the version of HPROF shipped with OpenJDK version 7, Yourkit 12.0.5 and JProfiler 8.0. As Free Lunch is implemented in Hotspot, we do not compare it with the four 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. We also present an experimental study of lock metrics on typical synchronization scenarios, an analysis of lock CSP for a set of more than 30 applications and a case study of a lock contention issue found in the Cassandra database. 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.

4.1   Profiler overhead

We compare the overhead of Free Lunch to that of HPROF, Yourkit and JProfiler running in lock profiling mode, on the 11 applications from the DaCapo 9.12 benchmark suite [12], the 19 applications from the SPECjvm2008 benchmark suite [87], and the SPECjbb2005 benchmark [86]. 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, Tradebeans does not run with Yourkit, and Avrora and Derby do not run with HPROF.

4.1.1   Overall performance results

Figure 4.1 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 4.1.a presents the complete

results, on a logarithmic scale, while Figure 4.1.b focuses on the case between 20% speedup and 60% slowdown.

Figure 4.1.a shows that the overhead of HPROF can be up to 4 times, that of Yourkit up to 1980 times and that of JProfiler up to 7 times. Figure 4.1.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.

4.1.2 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.

Supporting general-purpose profiling. HPROF, like the other existing profilers, is implemented outside the JVM, relying on JVMTI [55], 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
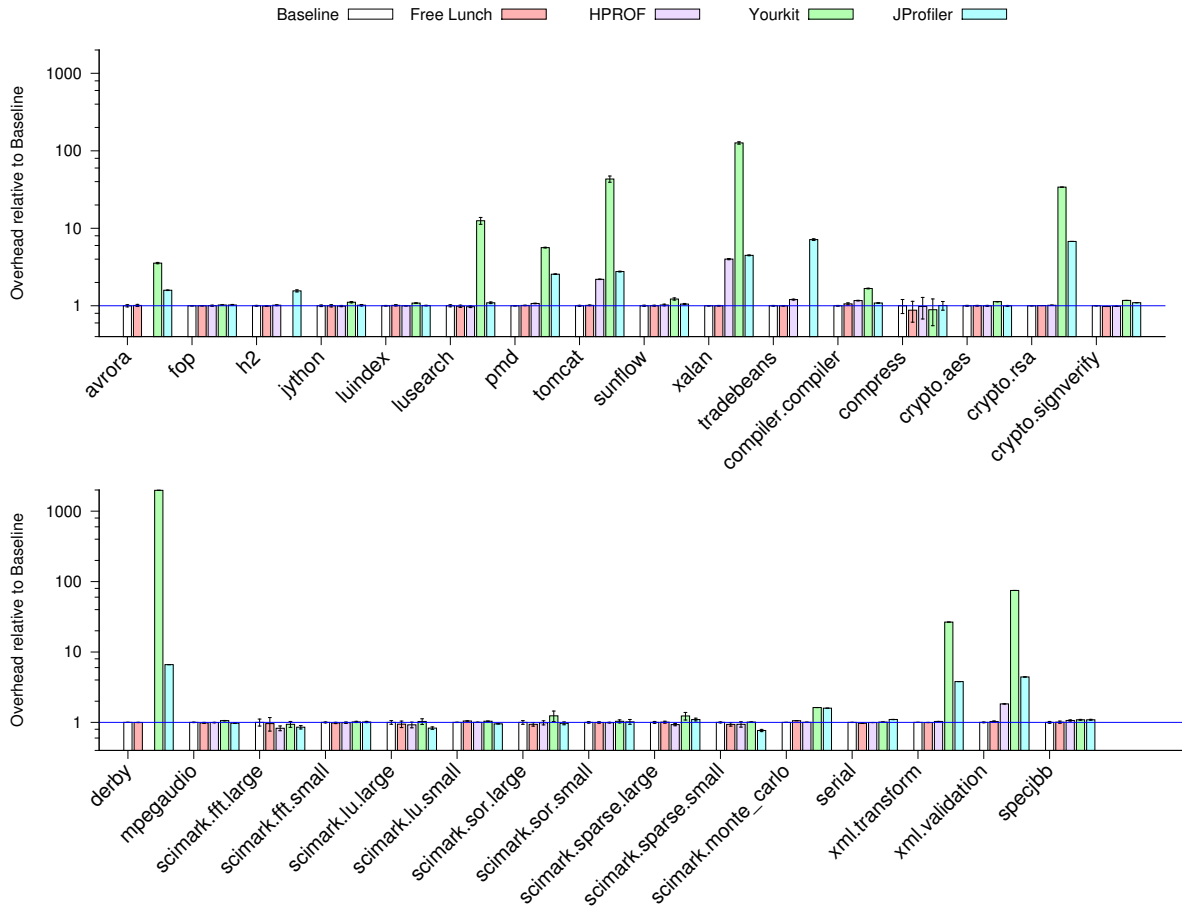
(a) Overhead on execution time with a logarithmic scale.
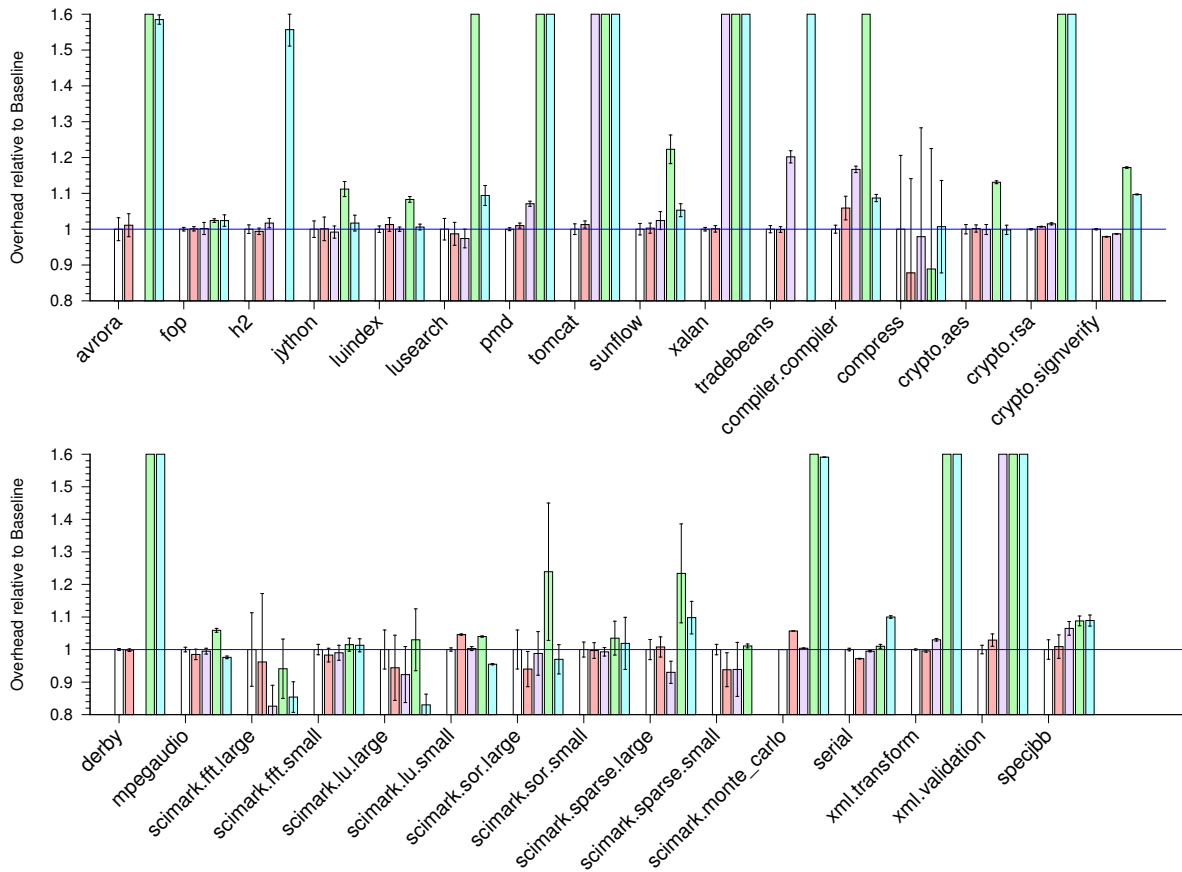
(b) Overhead on execution time, limited to between 80% and 160% (zoom of (a)).

Figure 4.1: Overhead on execution time compared to baseline.

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.

4.2 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.

4.2.1 OptHPROF

To make our baseline, OptHPROF, for comparison with Free Lunch, we remove the two main bottlenecks presented in Section 4.1.2. 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` [62].
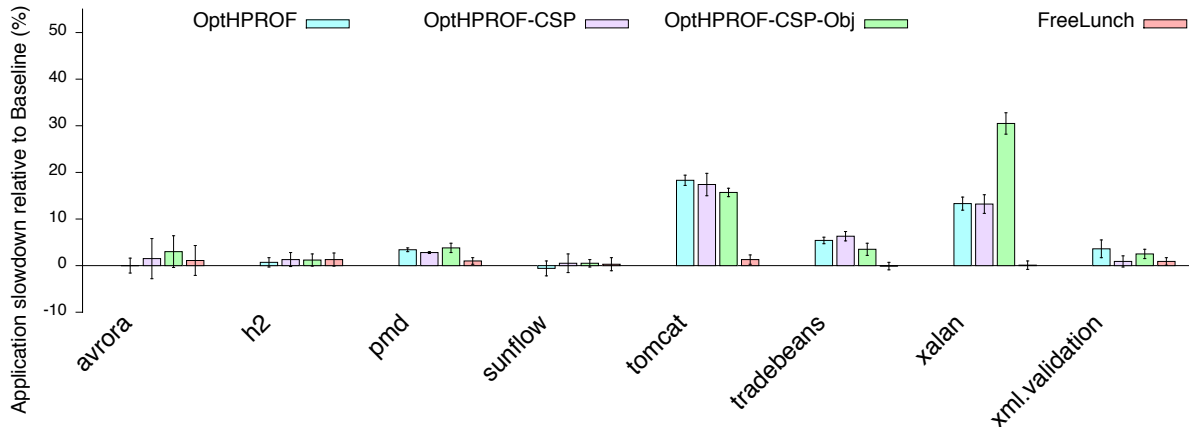
Figure 4.2: Overhead on execution time compared to baseline.

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 4.2 reports the overhead of OptHPROF on Avrora, H2, PMD, Sunflow, Tomcat, Tradebeans, Xalan and Xml.Validation, 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%.

4.2.2 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,

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 4.1: Experiments conducted to understand Free Lunch.

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.

We evaluate each of these differences in terms of the set of experiments described in Table 4.1. Each experiment involves creating a variant of OptHPROF that mimics Free Lunch in one or more of the above aspects, Figure 4.2 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 4.2, 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 4.2, 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 4.3 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 [64]. 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.
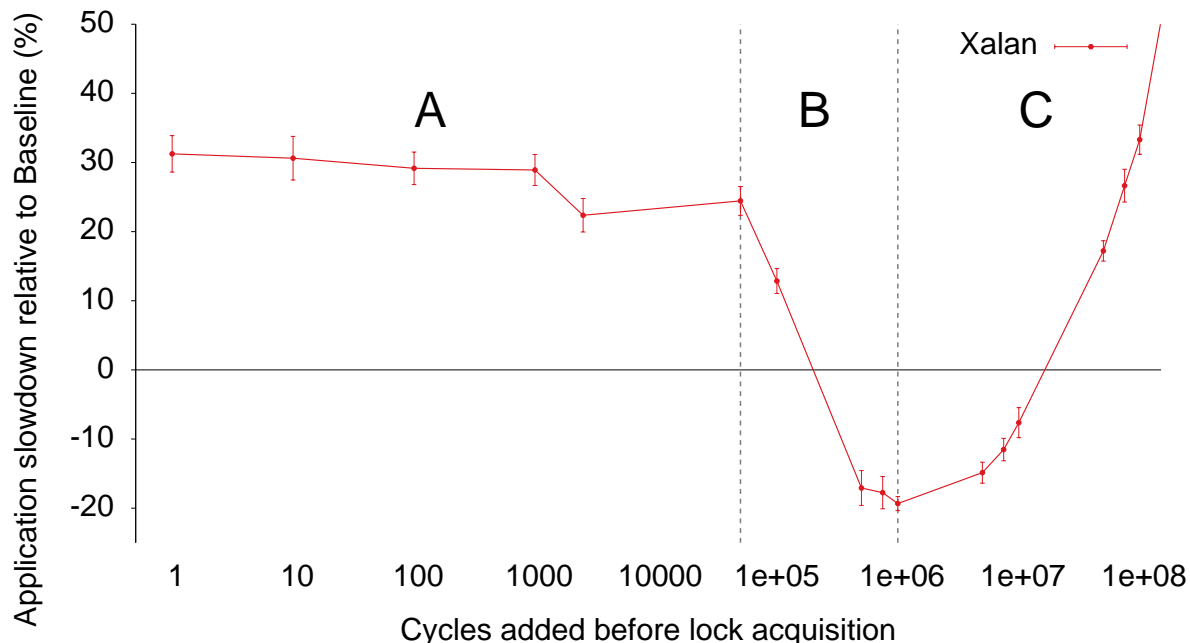
Figure 4.3: Overhead of Xalan with a varying delay before lock acquisition.

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 4.2, 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%.

4.3 Using Free Lunch to analyze applications

We now experimentally validate our analysis of the metrics presented in Section 3.1 and report our results when using Free Lunch to analyze the lock behavior of the applications considered in Section 4 as well as Cassandra 1.0.0 [61]. All evaluations are performed on the machine described in Section 4.

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 4.2: Evaluation of contention metrics on the ping-pong micro-benchmark.

4.3.1   Micro-benchmarks

We instantiate the scenarios described in Section 3.1 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 4.2.

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 4.2). 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 4.4.

For this micro-benchmark, we compare Free Lunch with Health Center, which relies on the number of failed acquisitions. As shown in Figure 4.4, the CSP reported by Free Lunch decreases with the processing time of the workers. On the other hand, the number of failures divided by
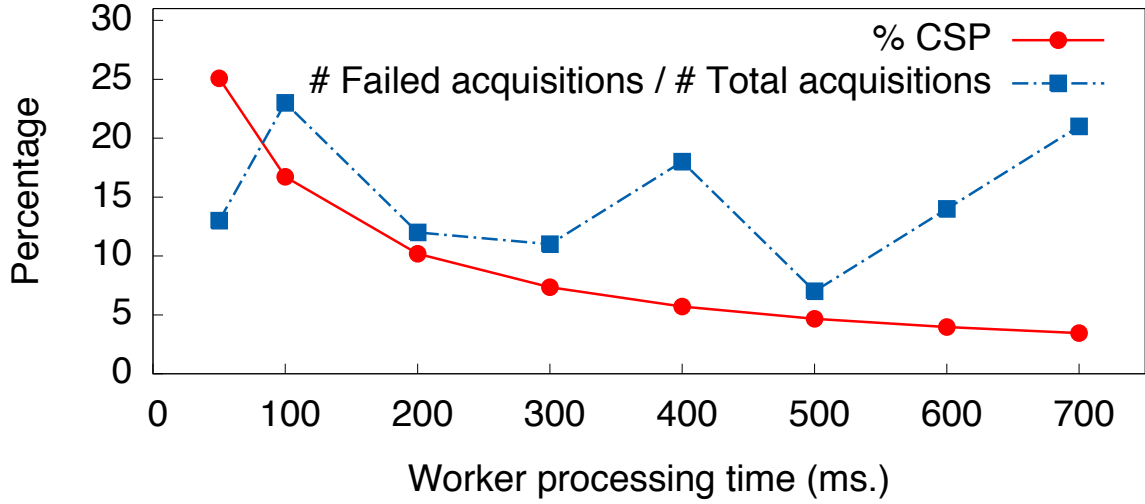
Figure 4.4: Comparison of the Free Lunch CSP metric and the Health Center metric on the fork-join micro-benchmark.

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 3.1: 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.

4.3.2   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 [12], the SPECjvm2008 [87] benchmark suite, and the SPECjbb2005 [86] 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 4.3 lists the locks with an average CSP greater than 5% in this case. Figure 4.5 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 4.3) is not equal to the average of the CSPs of each individual measurement interval (Figure 4.5), 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.

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 4.3: CSP averaged during the whole run.

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 processed sequentially; thus, the more clients send requests to the database, the more clients try to acquire the lock. As shown in Figure 4.5, 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 4.5, 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 4.5, 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 4.5, 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
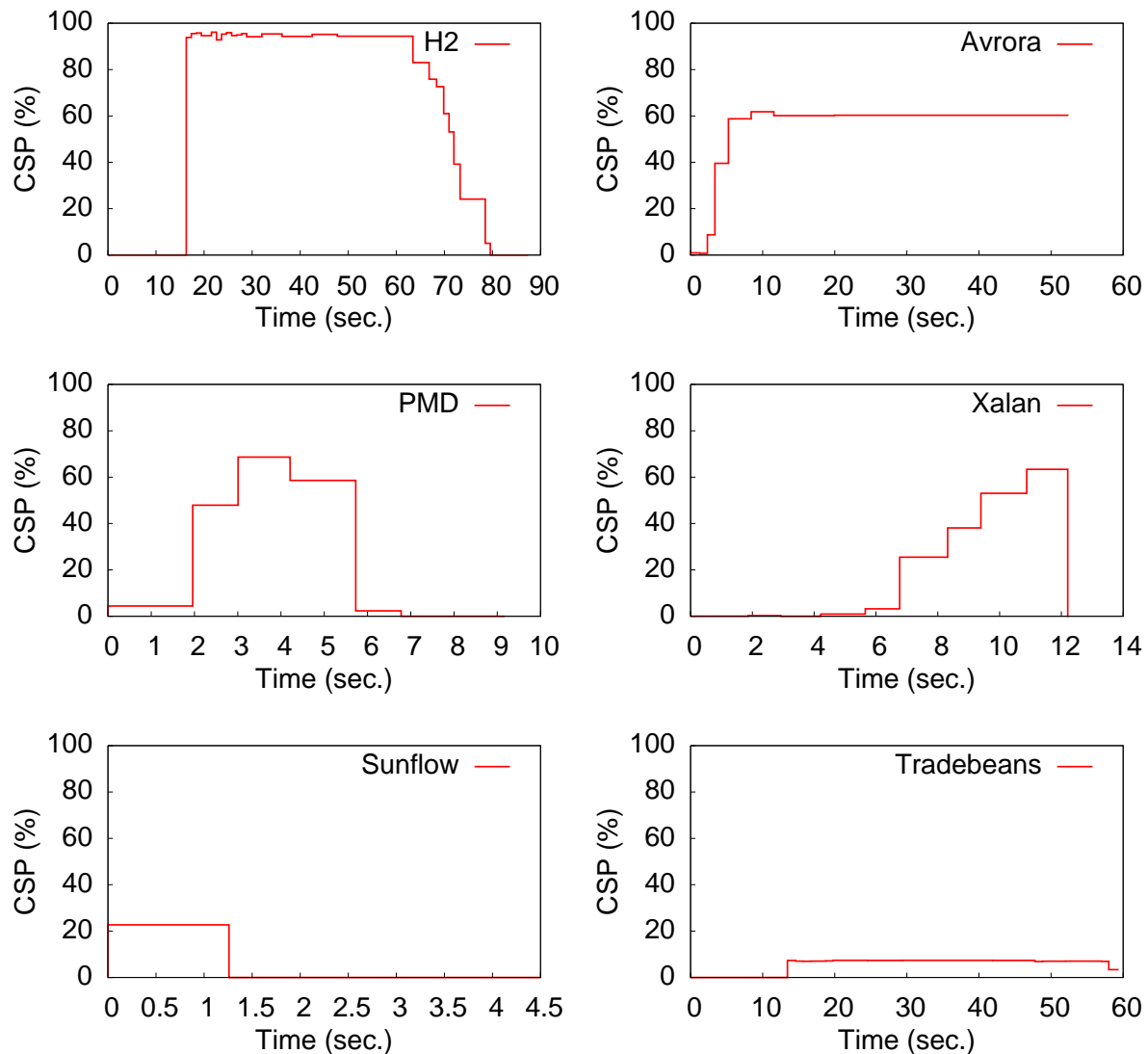
Figure 4.5: CSP with a measurement interval of 1s.

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 6.2%. As shown in Figure 4.5, 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 4.5, 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.

4.3.3 Cassandra

Cassandra [61] is a distributed on-disk NoSQL database, with an architecture based on Google's BigTable [16] and Amazon's Dynamo [27] 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.[1] 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 [77] 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

---

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

during *in-vitro* testing is small, and thus *in-vivo* profiling is essential.

Chapter 5

Conclusion

This thesis has presented Free Lunch, a new lock profiler especially designed to identify *phases* of high lock contention *in-vivo*. The role of Free Lunch is to identify locks that hamper application responsiveness and throughput, with an emphasis on servers class applications that typically leverage multicore hardware and for which these properties are crucial. We introduce a novel metric called Critical Section Pressure (CSP) that evaluates the lack of thread progress due to synchronization issues. The CSP is defined by the percentage of time spent by the threads in acquiring the lock over the cumulated running time of these threads. The CSP metric is implemented inside Free Lunch and helps to report back to developers the percentage of time during which threads are blocked and unable to make progress due to a lock. Free Lunch is also designed to compute regularly the CSP metric in order to detect phases of lock contention that may arise unexpectedly due to various external factors difficult to reproduce *in-vitro*. Free Lunch is designed in a way that limits the profiling overhead to what is acceptable for *in-vivo* profiling. The key idea in the implementation is that Free Lunch is embedded within the JVM, leveraging direct access to the locking subsystem and internal data structures in a flexible way and using an efficient time management facility. This approach is done detrimental to portability, nevertheless, Free Lunch is only 420 lines of code long which should be easy to implement in another JVM.

Free Lunch is evaluated on applications coming from the DaCapo 9.12, SPECjvm2008 and SPECjbb2005 benchmark suites, and the Cassandra database with a workload from YCSB for a total of 32 applications. The hardware used for these experimentation is a server with four AMD Opteron processors for a total of 48 cores at a 2.2GHz frequency, having 256GB of RAM and running a Linux 3.2.0 64-bit kernel from Ubuntu 12.04. We identified phases of high CSP in six applications. 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 also found a phase of high CSP in the Cassandra database. This high CSP phase happens during the replay of the commit log, performed during the recovery of a crashed node in order to return to a steady state. We studied existing metrics of state-of-the-art lock profilers theoretically and empirically against synchronization scenarios typically encountered in multithreaded applications. We found that these metrics do not return insightful

results to help developers to highlight locks that are harmful for the application. We 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*. We provide a detailed analysis of overhead costs associated with each design choice in HPROF to help understand the origin of its overhead. Then starting with baseline HPROF, we gradually improve it until we reach the Free Lunch final design.

Future work

Future work for Free Lunch will focused around addressing its limitations about the precision CSP computation and extending it to different locking mechanisms.

Free Lunch has no application-specific information about the role of the individual threads, and thus assumes that all threads are equally important to the notion of progress. For example, in the generalized ping-pong scenario, it may be that the two ping-pong threads control output to the user, and the remaining threads perform computations whose results will be ultimately discarded, if the ping-pong threads cannot output them sufficiently quickly for instance. A low CSP for this scenario would not reflect the user experience. It would require to have a deeper knowledge of application architecture to differentiate threads according to the task they perform and vary the importance of blocking time for the CSP computation accordingly. To achieve that, it could be possible to leverage techniques like category analysis as it is done for the WAIT tool [5] and for Capacity planning [68].

A limitation of our design is that Free Lunch only takes into account lock acquisition time as being detrimental to thread progress. However, Free Lunch does not record every situation where threads are prevented from progressing. Such situations include ad hoc synchronization [96] where developers write their own synchronization mechanism and bypass those offered by programming languages or locking APIs such as POSIX. In Java, apart from the *synchronized* keyword, threads could synchronize with volatile variables, lock-free algorithms or data structures [62].

Free Lunch exclusively considers synchronized blocks and methods for locking. Programs now make extensive use of the java.util.concurrent package [62] but Free Lunch does not profile it yet. To the best of our knowledge, only JProfiler and JUCProfiler (a tool from MSDK) has the ability to profile these locks. A similar work to what we have done on synchronized blocks could be carried on, namely to see if existing j.u.c metrics give insightful data for finding lock issues, assessing these metrics on similar micro-benchmarks and real applications, evaluating the overhead of these profilers to see if it is suitable for *in-vivo* profiling, and if needed, designing a better profiler for these locks. For Hotspot, the implementation will raise new challenges since it is a different locking subsystem, split between pure Java code in the JCL and C++ code located inside the JVM, as opposed to synchronized blocks which are completely implemented inside the JVM.

Later works that cite Free Lunch

Following the publication that presented the work around Free Lunch (in OOPSLA '14 [24] and ComPAS '13 [22], as well as the INRIA research report [23], the EuroSys 2012 Doctoral Workshop and a poster for the EIT ICT Labs Symposium on Future Cloud Computing in 2014), Free Lunch has been cited by Curtsinger et al. [20] in a work-in-progress paper that introduces casual profiling, an approach to identify exactly where programmers should focus their optimization efforts by virtually speeding up lines of code and quantifying its impact on throughput and latency.

Appendix A

French summary of the thesis
Synthèse du rapport de thèse en français

Following the rules of the Université Pierre et Marie Curie, this appendix is a short summary of the thesis, written in French. *Afin de suivre les règles de l'université Pierre et Marie Curie, cette annexe contient une synthèse du rapport de thèse en français.*

A.1   Introduction

Nous vivons dans un monde de données et notre production journalière de données augmente de façon exponentielle. En 2013, le consortium International Data Corporation (IDC) a estimé la taille de l'Univers Digital à environ 4,4 zettaoctets (4.4×10^{21}) et prévoit qu'il doublera tous les 2 ans pour atteindre 44 zettaoctets en 2020 [48]. Cette tendance est connue sous le nom de Big Data. De plus, le nombre d'objets connectés dans l'Internet des Objets, qui désigne les objets de la vie courante dotés d'une connexion à Internet leur donnant la possibilité de transmettre des données, a été estimé à environ 20 milliards d'appareils en 2013 et est prévu d'atteindre 32 milliards d'ici 2020, comptant pour 10 % de l'Univers Digital. Ces prévisions soulignent le fait que les données sont omniprésentes dans notre vie et que cette tendance va s'accentuer dans les années à venir.

Tirer profit des Big Data est un problème majeur pour des industries travaillant dans la finance, la technologie, la santé, la distribution ou l'énergie car cette technologie est considérée comme l'un des plus important vecteur de création de valeur pour l'avenir. Les Big Data peuvent aider les entreprises à prendre de meilleures décisions, par exemple, comprendre les habitudes de consommation de leurs clients, optimiser leurs processus opérationnels et de contrôle, pour prendre de meilleurs décisions pour le prix de vente de leurs produits et beaucoup d'autres. Quelques exemples d'applications utilisant les Big Data incluent l'outil Facebook Graph Search [19] qui permet d'effectuer des recherches multicritères avancées dans leur graphe d'utilisateurs afin de répondre à des requêtes complexes pour cibler les clients désirés ou les systèmes de recommandations utilisé par Youtube [25] ou Netflix [58, 59] où le système recommande des ensembles de vidéos personnalisés aux utilisateurs, basés sur leur activité sur le site. Cependant, cela reste un challenge de pouvoir structurer ces données de façon compréhensive, de les traiter avec une

faible latence, d'en extraire les informations appropriées et de reporter ces résultats aux clients.

Traiter ces énormes masses de données soulève d'importants challenges pour la communauté système. Les applications typiques de Big Data telles que les systèmes d'analyse à grande échelle comme MapReduce [26] ou Spark [99], les bases de données [40, 61] et les serveurs web [7, 50] ont d'importants besoins en terme de calculs et de mémoire et le temps de réponse et le débit sont critiques pour une expérience utilisateur optimale [76]. Ces programmes sont habituellement parallélisés et utilisent des serveurs multicœurs situés dans des data-centers comme plateforme de calcul. Cependant, la parallélisation des programmes est un problème notoirement difficile qui peut empêcher de tirer parti de toute la puissance de calcul de telles plateformes, en particulier à cause de la loi d'Amdahl [6, 36]. Cette loi stipule que le potentiel d'accélération obtenu en parallélisant un programme est limité par la partie séquentielle du programme. Les portions séquentielles d'un programme sont appelées les sections critiques: elles protègent les données partagées des multiples accès concurrents et sont entourés de verrous pour assurer la cohérence des données.

Cependant à cause de la complexité de ces applications, il se peut que certaines sections critiques ne soient pas efficientes dans toutes les configurations d'exécution. Ces sections critiques peuvent entraver l'avancement des threads dans des conditions spécifiques et peuvent dégrader de façon drastique le temps que met le serveur pour traiter les requêtes. Les développeurs essayent généralement de trouver ces sections critiques problématiques pendant la phase d'évaluation des performances mais cela ne permet pas toujours de toutes les identifier. Cela est dû à 3 principales raisons:

- La difficulté à reproduire un environnement d'exécution réel: le logiciel utilise très probablement un jeu de données représentatif de la charge de travail attendue pour les tests. Dans le meilleur des cas, les développeurs essayeront de simuler le jeu de données le plus représentatif, aussi proche que possible des conditions réelles d'utilisation, afin de tester l'application. Cependant, ce jeu de données est dépendant du cœur de métier utilisant le logiciel et sera ultimement généré par les utilisateurs, il est donc complètement inconnu avant le déploiement. Il est possible qu'il soit complètement différent de ce que les développeurs avaient envisagé, en particulier si le logiciel est assez flexible pour être utilisé dans une large variété de situations,

- La difficulté à reproduire tous les scénarios possibles d'exécutions: la charge de travail de test appliquée au logiciel est généralement composé d'un mélange de requêtes prédéfinies. Les requêtes des utilisateurs ne sont pas prévisibles à l'avance et elles exposent le logiciel à une multitude de requêtes différentes à traiter. Il est difficile de savoir comment stresser l'application avec une charge de travail proche de celle qu'elle rencontrera dans des conditions réelles d'utilisation,

- Impossibilité de tester toutes les configurations matérielles possibles: les développeurs ont généralement accès à un ensemble restreint de machines pour leurs tests. Ils ne peuvent pas évaluer les applications sur un large ensemble d'architectures et de processeurs où les résultats pourraient varier. Les performances peuvent aussi beaucoup varier entre différentes versions du même système d'exploitation ou de la machine virtuelle Java.

Parfois, ils testent leurs applications sur leur propre machine qui est loin de ressembler à un serveur avec un nombre important de cœurs et beaucoup de mémoire. Il n'est pas réaliste de tester toutes ces combinaisons et les développeurs finissent par tester seulement un sous-ensemble des architectures, des systèmes d'exploitation, des machines virtuelles et des différentes versions d'une application existantes.

Pour ces raisons et malgré des tests rigoureux, il est difficile de simuler tous les scénarios de façon exhaustive. Par conséquent, le débit et le temps de réponse peuvent être dégradés dans des situations qui n'étaient pas prévues par les développeurs pendant la phase de développement et qui seront découvertes au moment du déploiement dans des conditions réelles d'utilisation, avec des conséquences dommageables pour l'expérience utilisateur. Par exemple, la PDG de Google Marissa Meyers a signalé à la conférence Google I/O qu'une augmentation du temps de réponse d'une demi-seconde pouvait mener à une baisse de trafic de 20% [66], entraînant ainsi une baisse des revenus publicitaires.

Java est régulièrement utilisé pour implémenter ces applications multithreadés complexes. Ce langage est devenu l'un des plus utilisé grâce à sa sécurité, sa flexibilité et son environnement de développement mature [91]. Néanmoins, le langage Java est connu pour ne pas être adapté aux architectures multicœurs. La principale abstraction du langage pour gérer la concurrence est le mot-clé *synchronized* qui encourage l'utilisation de synchronisation à gros-grain. En dépit des efforts fait par la communauté Java avec par exemple, le package java.util.concurrent [62] qui a pour but d'offrir un ensemble d'abstractions à grain-fin pour le contrôle de la concurrence, les blocks synchronized restent très largement utilisés. Par exemple, il y a approximativement 7410 blocks synchronized situés dans la Java Class Library de Java 7. Les applications ne peuvent pas être optimisées finement pour être exécutées sur du matériel multicœurs spécifique, en prenant en compte par exemple le comportement des caches ou la hiérarchie mémoire car ces fonctionnalitées sont cachées par la machine virtuelle Java (JVM). Enfin, la formation et l'expérience des développeurs Java sont habituellement orientées vers des aspects logiciels de haut-niveau plutôt que vers des problèmes de synchronisation de bas-niveau.

De plus, un profilage effectif des applications Java pour serveur requiert l'utilisation d'une métrique qui reporte la dégradation des performances du serveur causée par un verrou et qui prend en compte le fait que ces applications ont un long temps d'exécution avec de multiples phases. Les logiciels de profilage de verrous pour Java reportent la contention moyenne pour chaque verrou par rapport à la durée totale d'exécution de l'application en utilisant de multiples métriques. Ces métriques se concentrent cependant sur l'identification des verrous les plus utilisés ou les plus contendus mais ne corrèlent pas ce résultat à l'avancement des threads, ce qui les rend incapables d'indiquer si un verrou représente un goulot d'étranglement ou non. Par exemple, sur un schéma de synchronisation classique tel que le fork-join, nous avons observé qu'un verrou fréquemment utilisé ou contendu n'entrave pas systématiquement l'avancement des threads. De plus, en reportant uniquement une moyenne par rapport à la durée totale d'exécution de l'application, ces profilers de verrous ne sont pas capables d'identifier les variations locales dues aux propriétés des différentes phases de l'application. Un verrou contendu pendant une phase peut nuire au temps de réponse mais il peut être masqué dans les résultats du profilage par une longue durée d'exécution de l'application.

Ces problématiques sont illustrées par un rapport de bug reporté 2 ans plus tôt dans la version 1.0.0 de la base de données distribuée NoSQL Cassandra [61][1]. Dans une configuration particulière, avec 3 nœuds et un facteur de réplication de 3, le temps de réponse de Cassandra est multiplié par 20 lorsqu'un nœud ne répond plus. Ce ralentissement est causé par un verrou utilisé dans l'implémentation du *hinted handoff*[2], un mécanisme qui enregistrent les transactions des nœuds dans le but de les rejouer plus tard lorsqu'un nœud revient en ligne après une panne. Il semble que les développeurs n'aient pas pensé à tester ce scénario spécifique ou qu'ils l'aient testé mais qu'ils n'aient pas été capable de reproduire le problème. De plus, même si ce scénario avait été exécuté, les profilers de verrous actuels auraient été incapable d'identifier la cause du goulot d'étranglement si le scénario avait été activé pendant une longue durée d'exécution qui aurait masqué la phase de contention.

Les recherches conduites dans cette thèse se concentrent sur la thématique du profilage de verrous et plus précisément, sur le problème de la dégradation des performances des applications Java pour serveurs due aux verrous sur des architectures multicœurs. Pour les raisons présentées précédemment, nous avons conçu un profiler de verrous doté des propriétés suivantes:

1. Le profiler doit utiliser une métrique qui indique si un verrou entrave l'avancement des threads. Le rapport du profilage à l'intention du développeur doit donner un aperçu clair et précis à propos de l'impact que les verrous ont sur la performance des applications, en particulier en terme de temps de réponse et de débit. Cela permettra au développeur de concentrer ses efforts de débogage sur un problème qui diminue réellement les performances de l'application,

2. Le profiler doit recalculer cette métrique périodiquement afin d'être sensible aux différentes phases d'une application. Les applications de type serveur sont complexes et leur comportement est dépendant de nombreux facteurs tels que un environnement imprévisible, des pics de charge à plusieurs moments dans la journée, une large variété de requêtes et le comportement des clients, tout cela n'étant pas prévisible théoriquement. Tous ces scénarios ne peuvent pas être anticipés dans un environnement de test et par conséquent, il est difficile de détecter tous les problèmes de verrous. Un profiler calculant et reportant régulièrement une métrique évaluant la contention des verrous permettra de trouver les problèmes liés aux particularités des clients et de l'environnement,

3. Le profiler doit induire un faible surcoût d'exécution pour être utilisé *in-vivo*. Un profiler *in-vivo* surveille continuellement l'application pendant son exécution. Cependant les utilisateurs ne sont pas prêts à à utiliser un profiler qui dégrade les performances de leur application de façon drastique. Intuitivement, il est également contradictoire de ralentir une application continuellement dans le but de trouver un hypothétique problème qui réduirait les performances de l'application. Par conséquent, le surcoût d'exécution du profiler doit être aussi limité que possible afin que ce ne soit pas détectable par le client final.

Dans cette thèse, nous proposons un nouveau profiler, appelé Free Lunch, conçu autour d'une nouvelle métrique, la Critical Section Pressure (CSP). Cette métrique a pour but d'évaluer

---

[1] https://issues.apache.org/jira/browse/CASSANDRA-3386.
[2] http://wiki.apache.org/cassandra/HintedHandoff.

l'impact de la contention des verrous sur l'avancement global des threads. La CSP est définie comme étant le pourcentage de temps passé par les threads de l'application à être bloqués en tentant d'acquérir un verrou pendant un intervalle de temps, ce qui indique le pourcentage de temps où les threads sont incapables de progresser et la perte potentielle de performance. Free Lunch est conçu spécifiquement pour identifier des *phases* où la CSP est élevée *in-vivo*: l'application est échantillonnée continuellement sur plusieurs intervalles de temps pendant lesquels la CSP est calculée pour chaque verrou. Quand la CSP d'un verrou atteint un seuil prédéfini, Free Lunch reporte l'identité du verrou aux développeurs avec une trace d'appels menant à la section critique protégée par le verrou, comme pour les applications et les systèmes d'applications qui retournent des rapports d'erreurs aux développeurs lors d'une panne ou d'une situation inattendue [38].

Afin de faire en sorte que le profilage *in-vivo* soit acceptable, Free Lunch doit induire un faible surcoût d'exécution. Pour réduire le surcoût d'exécution, Free Lunch tire parti des structures de données des verrous internes à la JVM en leurs ajoutant une structure de données additionnelle contenant les données de profilage. Ces structures de verrous sont déjà protégées de la concurrence d'accès et donc Free Lunch ne nécessite aucune synchronisation additionnelle pour accéder aux données de profilage. Free Lunch ajoute le calcul périodique de la CSP dans le système de gestion des verrous de la JVM afin d'éviter des inspections supplémentaires des threads ou des verrous. Free Lunch repose également sur des instructions matérielles spécifiques fournissant des fonctionnalitées de gestion du temps efficace permettant une instrumentation minimale du code en charge du verrouillage. Grâce à cela, Free Lunch ajoute seulement 11 instructions assembleurs dans la fonction d'acquisition des verrous sur une architecture amd64.

Nous avons implémenté Free Lunch dans la JVM Hotspot 7 [89]. Cette implémentation modifie seulement 420 lignes de code, majoritairement dans le sous-système de gestion du verrouillage, ce qui laisse penser qu'il devrait être aisé de l'implémenter dans une autre JVM. Nous comparons Free Lunch avec d'autres profilers sur une machine AMD Magny-Cours de 48 cœurs en terme de performance et d'utilité des résultats du profilage. Nos principales contributions sont les suivantes:

- Théoriquement et expérimentalement, nous avons trouvé que les métriques pour évaluer la contention des verrous utilisé par les profilers Java existants HPROF [42], JProfiler [52], Yourkit [97], MSDK [69], IBM Health Center [41], Java Lock Monitor [67] et Java Lock Analyzer [49] sont inappropriées pour identifier si un verrou entrave l'avancement des threads.

- Free Lunch permet de détecter une phase précédemment non signalée avec une CSP élevée dans le sous-système de rejeu des transactions de Cassandra. Ce problème est resté invisible aux développeurs de Cassandra car il est déclenché par un scénario particulier et survient uniquement pendant une courte période durant l'exécution, ce qui le rend difficile à détecter avec les profilers existants.

- Free Lunch a permis d'identifier 6 verrous ayant une CSP élevée dans 6 applications provenant de benchmarks standards. À partir de ces résultats, nous avons amélioré les performances de l'une de ces applications (Xalan) de 15 % en changeant une seule ligne de code. Comme le verrou est contendu seulement pendant la moitié du temps total d'exécution de

Table A.1: Capacité des métriques à évaluer l'avancement des threads.

Métriques	Scenario		Profilers
	ping-pong	fork-join	
# acquisitions échouées / # acquisitions	+	-	JLM, JLA, Health Center
# acquisitions échouées / Temps d'exécution	-	-	JLM, JLA, MSDK, Health Center
Temps total en section critique / # acquisitions	-	+	JLM, JLA, MSDK, Health Center
Temps d'acquisition d'un verrou / Temps d'acquisition de tous les verrous	-	-	HPROF
Temps d'acquisition d'un verrou / Temps d'exécution	-	-	HPROF, JProfiler, Yourkit

l'application, les autres profilers sous-estiment largement son impact sur les performances. Pour les autres applications, les informations retournées par Free Lunch nous ont aidés à vérifier que le comportement de verrouillage n'entravait pas assez l'avancement des threads pour avoir un impact significatif sur les performances de l'application ou bien que la section critique en question ne pouvait pas être facilement modifiée.

- Dans la suite de benchmarks DaCapo 9.12 [12], la suite de benchmarks SPECjvm2008 [87] et le benchmark SPECjbb2005 [86], nous avons trouvé qu'il n'y a aucune application pour laquelle le surcoût d'exécution moyen de Free Lunch était plus important que 6 %. Ce résultat montre qu'un profiler mesurant la CSP peut avoir des performances acceptables pour du profilage *in-vivo*.

- Les profilers de verrous compatibles avec Hotspot, HPROF [42], JProfiler [52] and Yourkit [97] induisent un surcoût d'exécution d'au maximum 4 fois, 7 fois et 1980 fois respectivement sur le même ensembles de benchmarks. Ces performances sont inacceptables pour du profilage *in-vivo*.

A.2   Conception du profiler Free Lunch

Le but de Free Lunch est d'identifier les verrous qui entravent le plus l'avancement des threads et de régulièrement mesurer l'impact des verrous sur l'avancement des threads au fil du temps. Nous décrivons dans cette section nos choix en ce qui concerne le design de Free Lunch par rapport à la définition de notre métrique, la durée de l'intervalle de mesure, les informations que Free Lunch reporte au développeur et les limitations de notre design.

A.2.1   La métrique Critical Section Pressure

En concevant une métrique dont le but est d'évaluer l'avancement des threads, nous observons tout d'abord qu'un thread est incapable de progresser quand il bloque pendant l'acquisition d'un

verrou. Cependant, prendre en compte seulement ce temps d'acquisition n'est pas suffisant: HPROF, Yourkit et JProfiler utilisent également le temps d'acquisition mais que les métriques qui en résultent sont incapables d'indiquer si un verrou entrave réellement l'avancement des threads (voir Table A.1). Notre proposition est de relier le temps d'acquisition d'un verrou au temps d'exécution cumulé des threads en définissant la *CSP d'un verrou* comme le ratio i) du temps passé par les threads à acquérir le verrou et ii) du temps d'exécution cumulé de tous les threads.

Afin de rendre cette définition plus précise, nous avons besoin de définir le temps d'exécution et le temps d'acquisition d'un thread, en considérant en particulier comment prendre en compte les cas où le thread est bloqué ou préempté pour des raisons diverses. Le temps où un thread attend sur une variable conditionnelle est exclu du temps d'exécution, par exemple comme dans les programmes Java où un thread attend sur une variable conditionnelle quand il n'a rien à faire. Cette observation est particulièrement vraie pour un serveur qui crée un large ensemble de threads pour traiter des requêtes mais où seule une faible portion de ces threads sont actifs à un moment donné. Ce temps d'attente n'est par conséquent pas essentiel pour le calcul de l'application et le prendre en compte réduirait drastiquement la CSP et rendrait difficile l'identification des phases pendant lesquelles les threads ne progressent pas. Par contre, le temps où un thread est bloqué pour d'autres raisons est inclus dans le temps d'exécution. Par exemple, considérons une application qui passe la majeure partie de son temps à faire des E/S en dehors de toute section critique et qui bloque rarement pour acquérir un verrou. Si le temps d'E/S n'est pas considéré, la CSP reportée sera élevée, même si le verrou n'est pas le goulot d'étranglement. De la même manière, dans un scénario opposé avec une application qui passe beaucoup de temps bloqué dans des E/S pendant qu'un verrou est détenu, ne pas compter les E/S pourrait mener à sous-estimer la CSP. Pour finir, le temps où les threads sont préemptés est inclus dans le temps d'acquisition et le temps d'exécution. La probabilité d'être préempté pendant l'acquisition d'un verrou est la même que la probabilité d'être préempté à n'importe quel autre moment pendant l'exécution, par conséquent, cela n'a pas d'impact sur le ratio entre le temps d'acquisition et le temps d'exécution cumulé des threads.

A.2.2 Intervalle de mesure

Afin d'identifier les phases de CSP élevée d'une application, Free Lunch calcule la CSP de chaque verrou pendant un intervalle de mesure. La calibration de la durée de cet intervalle de mesure doit prendre en compte 2 contraintes opposées. D'un côté, l'intervalle de mesure doit être assez court pour permettre d'identifier les phases d'une application. Si l'intervalle de mesure est long comparé à la durée d'une phase pendant laquelle il y a une CSP élevée, la CSP mesurée sera négligeable et Free Lunch sera incapable d'identifier cette phase de CSP élevée. Mais d'un autre côté, si l'intervalle de mesure est trop court, la présence de quelques threads bloqués durant l'intervalle peut donner une CSP élevée même s'il y a peu de pression sur les sections critique. Dans ce cas, Free Lunch identifiera de nombreuses phases avec une CSP élevée, masquant les véritables phases ayant une CSP élevée en reportant de nombreux faux-positifs.

Nous avons testé plusieurs durées pour l'intervalle de mesure en utilisant l'application Xalan de la suite de benchmarks DaCapo 9.12. Cette application est un parser XSLT transformant des
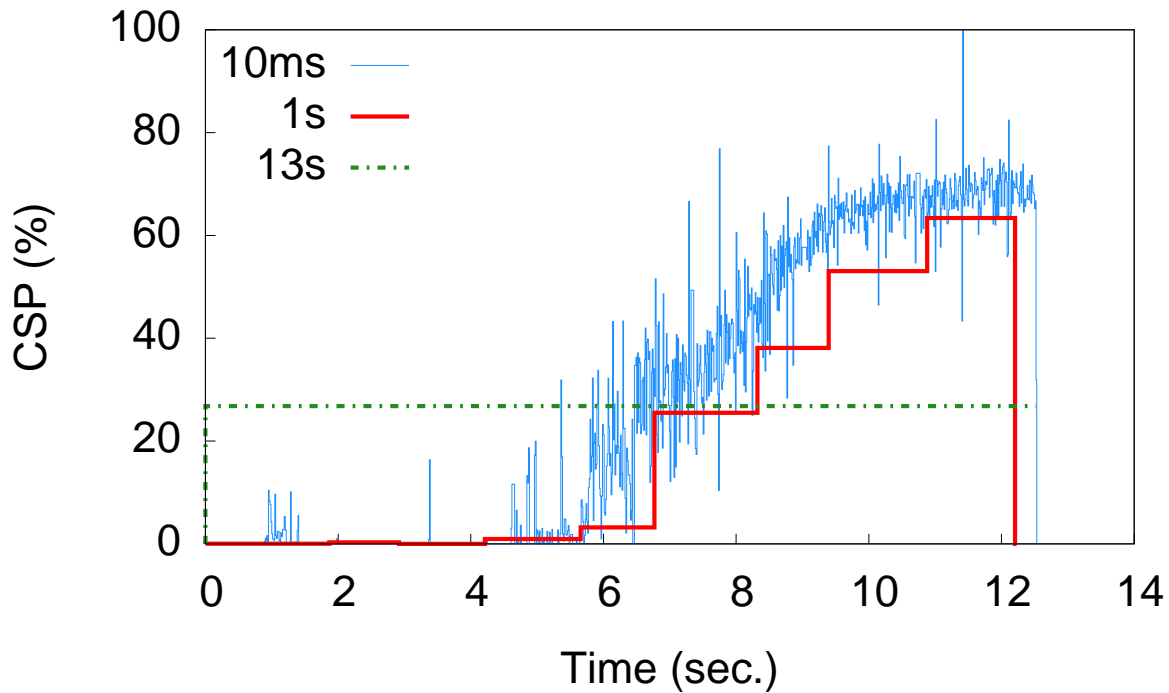
Figure A.1: CSP en fonction de l'intervalle minimal de mesure pour l'application Xalan.

documents XML en HTML. Xalan présente une phase avec une CSP élevée pendant la deuxième moitié de son exécution causée par de nombreux accès concurrents à une table de hachage protégée par un verrou. La Figure A.1 reporte l'évolution de la CSP pendant l'exécution. Avec un intervalle de mesure court de 10 ms., la CSP varie beaucoup entre les points de mesure successifs. Dans ce cas, le verrou va et vient entre un état contendu (avec une CSP élevée) et un état peu contendu (avec une CSP faible). À l'opposé, quand l'intervalle de mesure est approximativement égal au temps d'exécution (13 sec.), la CSP est moyennée sur la durée totale d'exécution ce qui masque les phases. Avec un intervalle de mesure de 1 sec., nous observons que (i) l'application a une CSP élevée pendant la deuxième moitié de l'exécution avec une valeur atteignant 64 % et que (ii) la CSP reste relativement stable entre 2 intervalles de mesure.

À partir de cette expérimentation, nous concluons qu'une seconde est un bon compromis car cet intervalle de mesure est assez long pour stabiliser la valeur de la CSP. De plus, si une phase avec une CSP élevée est plus courte qu'une seconde, il est possible que l'utilisateur ne remarque aucune dégradation du temps de réponse.

A.2.3 Rapport reporté par Free Lunch

Pour pouvoir aider efficacement les développeurs à identifier la cause d'une CSP élevée, Free Lunch reporte non seulement l'identité du verrou affecté mais également une trace d'appels menant à son acquisition. Free Lunch obtient cette information en traversant la pile d'exécution. Étant donné que traverser la pile d'exécution est coûteux, nous avons décidé d'enregistrer une

unique trace d'appels, celle menant à la première acquisition du verrou alors que celui-ci est déjà verrouillé par un autre thread. Des travaux précédents [5] et notre expérience dans l'analyse des programmes Java décrite dans la Section 4.3.2 montre qu'une seule trace d'appels est généralement suffisante pour comprendre pourquoi un verrou entrave l'avancement des threads.

A.2.4 Limitations de notre design

Une limitation de notre design est que Free Lunch prend uniquement en compte le temps d'acquisition des verrous comme étant un problème pour l'avancement des threads. De plus, il pourrait reporter une CSP faible dans le cas où les verrous sont rarement utilisés mais où beaucoup de threads ne peuvent progresser à cause de schémas de synchronisation ad-hoc [96] ou d'utilisation d'algorithmes non-bloquants [62].

De plus, Free Lunch n'a pas d'informations spécifiques provenant des applications à propos du rôle des threads. Par conséquent, il suppose que tous les threads sont aussi importants les uns que les autres pour la notion d'avancement. Par exemple, il se peut que 2 threads acquérant à tour de rôle un verrou contrôlent l'envoi du résultat vers l'utilisateur tandis que les threads restants effectuent des calculs dont les résultats seront au final inutilisés si les 2 threads ne peuvent pas les envoyer suffisamment rapidement. Une CSP faible pour ce scénario ne traduirait pas correctement l'expérience utilisateur.

A.3 Évaluation

Nous évaluons maintenant la performance de Free Lunch comparée aux profilers existants pour OpenJDK: la version de HPROF livré avec OpenJDK version 7, Yourkit 12.0.5 et JProfiler 8.0. Comme Free Lunch est implémenté dans Hotspot, nous ne le comparons pas avec les 4 autres profilers pour la JVM J9 d'IBM car Hotspot et J9 ont des performances non comparables. Nous comparons d'abord le surcoût d'exécution de Free Lunch par rapport à celui des autres profilers. Nous présentons ensuite une analyse de profilage pour un ensemble de plus de 30 applications et pour un cas d'étude sur un bug de performance trouvé dans la base de données Cassandra. Ce résumé n'inclut pas l'étude du coût de chaque choix de conception de Free Lunch ni l'étude expérimentale des métriques des profilers existants dans des scénarios typiques de synchronisation. Toutes nos expérimentations ont été effectuées sur un serveur doté de 48 cœurs AMD Magny-Cours cadencés à 2.2Ghz avec 256 Go de mémoire vive. Le système tourne sous le noyau Linux version 3.2.0 64-bit provenant de la distribution Ubuntu 12.04.

A.3.1 Surcoût d'exécution des profilers

Nous comparons le surcoût d'exécution de Free Lunch avec celui de HPROF, Yourkit et JProfiler en utilisant leur mode de profilage pour verrous sur les 11 applications de la suite de benchmarks DaCapo 9.12 [12], les 19 applications de la suite de benchmarks SPECjvm2008 [87] et le benchmark SPECjbb2005 [86]. Pour DaCapo, nous effectuons 20 exécutions de chaque application, avec 10 itérations par exécution, et prenons la moyenne du temps d'exécution de la dernière itération pour chaque exécution. Pour SPECjvm2008, nous configurons chaque application afin

qu'elle exécute 120 sec. du benchmark pour ne pas prendre en compte le temps d'initialisation de la JVM, suivi de 20 itérations de 240 sec. chacune. Pour SPECjbb2005, nous exécutons 20 fois une expérience qui utilise 48 "warehouses" et s'exécute pendant 240 sec., avec au préalable 120 sec. d'exécution du benchmark afin de ne pas prendre en compte le temps d'initialisation de la JVM. Pour SPECjvm2008 et SPECjbb2005, nous reportons le taux moyen d'opérations complétées par minute. Nous notons que quelques benchmarks n'ont pas pu être exécutés par certains profilers: H2 ne fonctionne pas avec Yourkit, Tradebeans ne fonctionne pas avec Yourkit, et Avrora et Derby ne fonctionnent pas avec HPROF.

La Figure A.2 présente le surcoût d'exécution induit par chaque profiler par rapport à la version originale d'Hotspot sans profilage, ainsi que l'écart-type autour de cette valeur. Les résultats sont présentés de 2 manières différentes à cause de leurs variations importantes. La Figure A.2.a présente les résultats complets sur une échelle logarithmique, tandis que la Figure A.2.b présente les résultats compris entre une accélération de 20 % et un ralentissement de 60 %.

La Figure A.2.a montre que le surcoût d'exécution de HPROF peut aller jusqu'à 4 fois, celui de Yourkit jusqu'à 1980 fois et celui de JProfiler jusqu'à 7 fois. La Figure A.2.b montre que pour toutes les applications, le surcoût d'exécution moyen de Free Lunch est toujours inférieur à 6 %. Pour quelques applications, l'utilisation du profiler semble améliorer les performances. Ces résultats ne sont pas concluants à cause d'un écart-type important.

Dans une configuration multicœurs comme ici, une des sources fréquentes de surcoût d'exécution est le problème de passage à l'échelle. Afin d'évaluer l'impact du passage à l'échelle du profilage, nous réalisons des expérimentations additionnelles avec HPROF, le profiler ayant le plus faible surcoût d'exécution maximum des profilers testés. Nous comparons HPROF à Hotspot sans le profilage sur le benchmark Xalan dans 2 configurations: l'une avec 2 threads sur 2 cœurs et l'autre avec 48 threads sur 2 cœurs. Dans les 2 cas, le surcoût d'exécution causé par le profiler est situé autour de 2 %, ce qui prouve que quand le nombre de cœurs est faible, le nombre de threads a un impact marginal sur les performances du profiler. Ensuite, nous effectuons ce même test avec Xalan et 48 threads sur 48 cœurs. Dans ce cas, Xalan termine en 4 fois plus de temps. Ces résultats suggèrent que, au moins dans le cas de HPROF, le surcoût d'exécution dépend principalement du nombre de cœurs.

A.3.2 Utilisation de Free Lunch pour analyser des applications

A.3.2.1 Analyse de la CSP

Cette section présente une analyse détaillée de la CSP des verrous utilisés par les applications de la suite de benchmarks DaCapo 9.12 [12], de la suite de benchmarks SPECjvm2008 [87] et du benchmark SPECjbb2005 [86]. Nous considérons d'abord le cas où l'intervalle de mesure est égal au temps d'exécution de l'application, ce qui représente la CSP moyenne de l'exécution. La Table A.2 liste les verrous ayant une CSP moyenne d'au moins 5 %. Ensuite, la Figure A.3 présente l'évolution de la CSP de ces mêmes verrous avec un intervalle de mesure d'une seconde. Notons que la CSP moyenne pendant toute la durée de l'exécution (Table A.2) n'est pas égale à la moyenne des CSPs de chaque intervalle de mesure individuel (Figure A.3) à cause du nombre de threads fluctuant entre dans chaque intervalle. Par exemple, une CSP élevée avec seulement 2
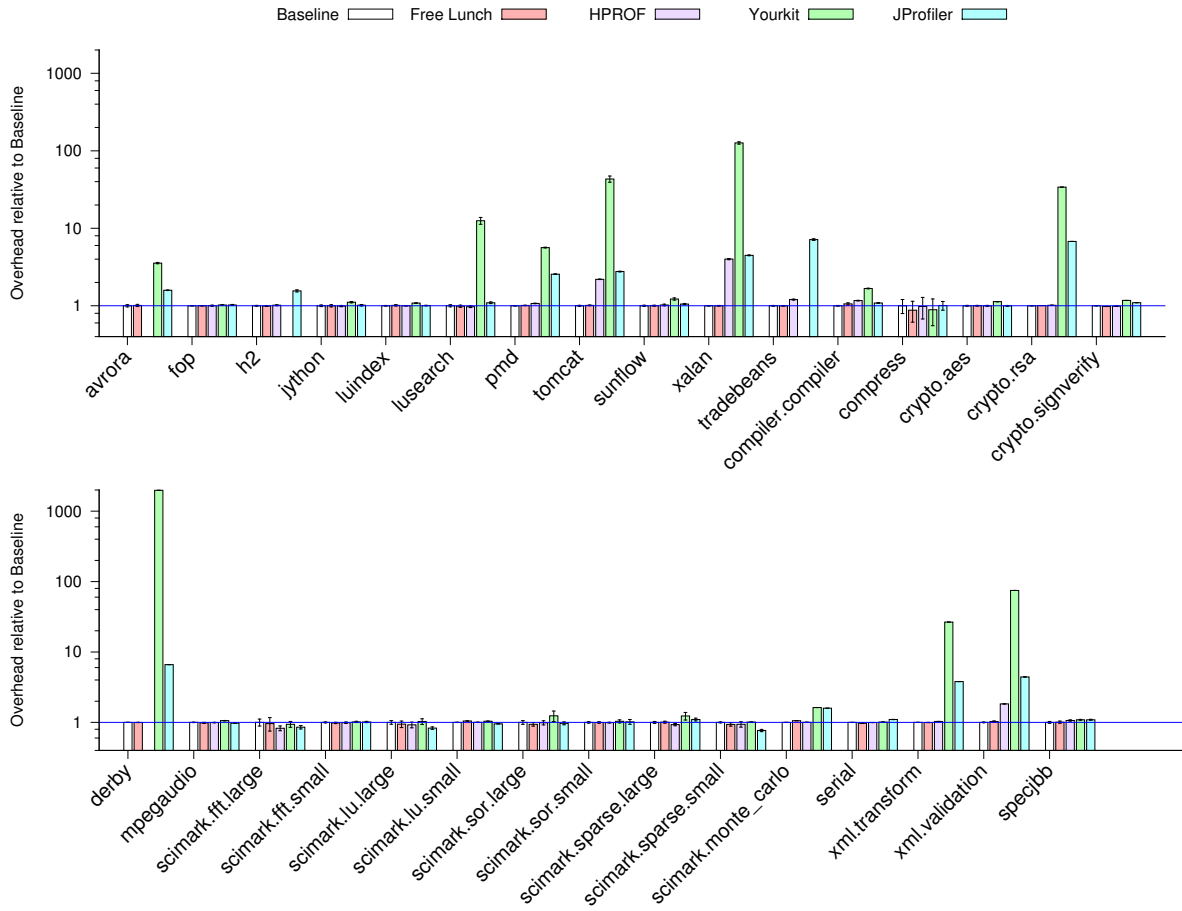
(a) Surcoût d'exécution des profilers sur une échelle logarithmique.
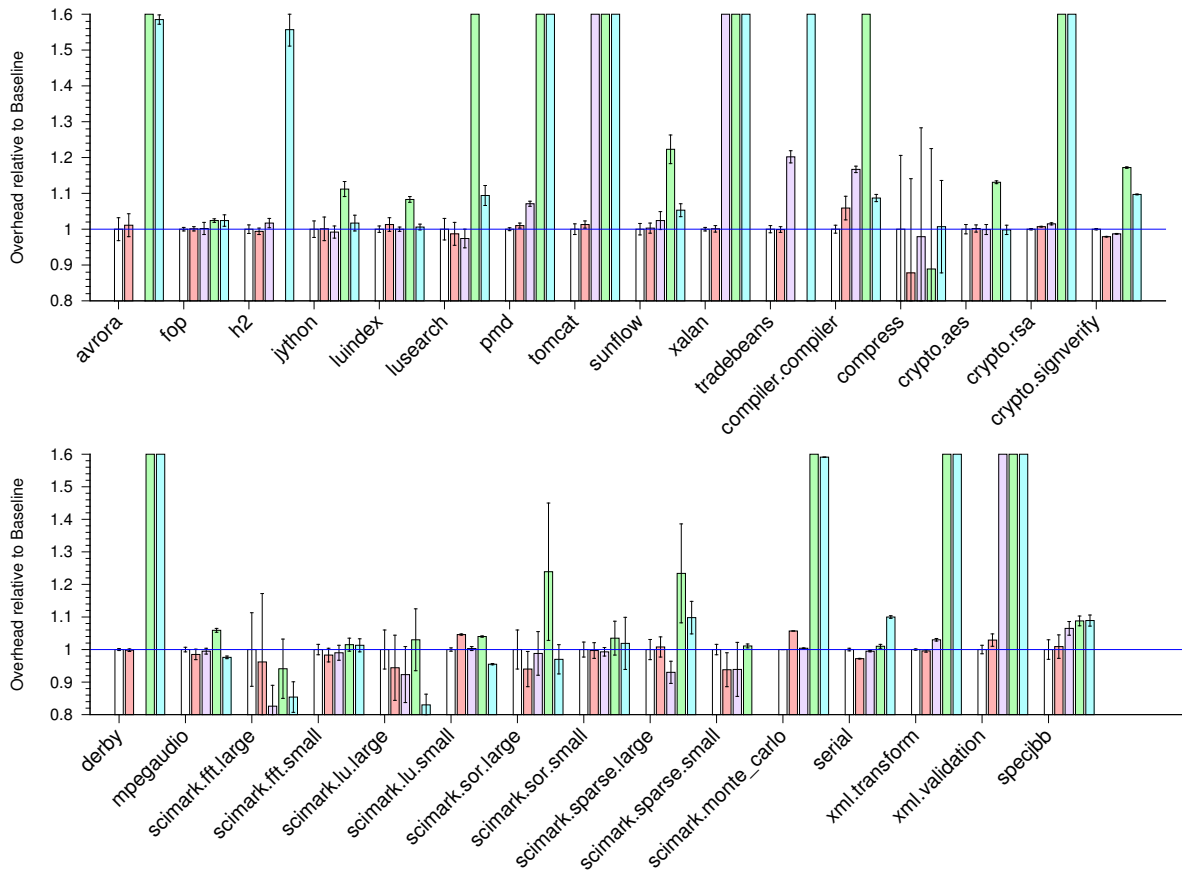
(b) Surcoût d'exécution des profilers, limité à des valeurs entre 80% et 160% (zoom de (a)).

Figure A.2: Surcoût d'exécution des profilers.

threads s'exécutant pendant un intervalle de mesure devient négligeable quand elle est moyenné sur 2 intervalles de mesure avec plusieurs threads s'exécutant dans le deuxième intervalle. Le reste de cette section analyse en détail la CSP de ces verrous.

Benchmark	Classe Java de l'objet avec la CSP la + élevée	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 A.2: CSP moyenne pendant l'exécution complète.

H2 est une base de données en mémoire. Le verrou associé avec l'objet `org.h2.Database` a une CSP moyenne de 62,3 %. H2 utilise ce verrou pour assurer que les requêtes des clients sont traitées séquentiellement, par conséquent, plus il y a de clients envoyant des requêtes à la base de données, plus les clients essayeront d'acquérir le verrou. Comme présenté dans la Figure A.3, H2 présente 3 phases distinctes. La première phase (de 0 sec. à 16 sec.) présente une CSP inexistante: dans cette phase, le thread principal de l'application remplit la base de données, par conséquent aucun verrou n'est contendu lors de l'accès à la base de données. La seconde phase (de 16 sec. à 79 sec.) présente une CSP entre 92 % et 96 %: les clients envoient les requêtes à la base de données, ce qui induit de la contention sur le verrou de la base de données. La CSP diminue à la fin de cette phase, passant de 92 % à 0 % quand les clients terminent leur requêtes à la base de données. Le but de cette dernière phase (de 79 sec. à la fin) est de revenir à l'état original de la base de données, ce qui est également effectué uniquement par le thread principal qui par conséquent n'induit pas de CSP. Cette application est fondamentalement impossible à faire passer à l'échelle car les requêtes sont traitées séquentiellement. Il serait nécessaire d'effectuer de profondes modifications de l'application pour améliorer les performances.

Avrora est un système de simulation et d'analyse. Le verrou associé avec l'objet `java.-lang.Class` a une CSP moyenne de 48,4 %. Avrora utilise ce verrou pour assurer la cohérence lors de l'obtention du résultat final. Comme présenté dans la Figure A.3, Avrora présente une phase de CSP élevée (de 2,3 sec. à la fin) où les threads applicatifs écrivent le résultat dans un fichier. Il semble qu'il n'y ait pas de solution simple pour enlever ce verrou car l'entrelacement des résultats des différents threads mènerait à un résultat incohérent.

PMD est un analyseur de code source. Le verrou associé avec l'objet `org.dacapo.harness.-DacapoClassLoader` a une CSP moyenne de 25,4 %. Cet objet est utilisé pour charger les nouvelles classes dynamiquement pendant l'exécution. Comme présenté dans la Figure A.3, une phase avec une CSP élevée démarre à 2 sec. et s'achève à 5,7 sec. alors que l'application se termine à 9,2 sec.. Pendant cette phase de CSP élevée, PMD stimule le chargeur de classe car tous les threads essayent de charger les mêmes classes. Il est probable que supprimer ce goulot
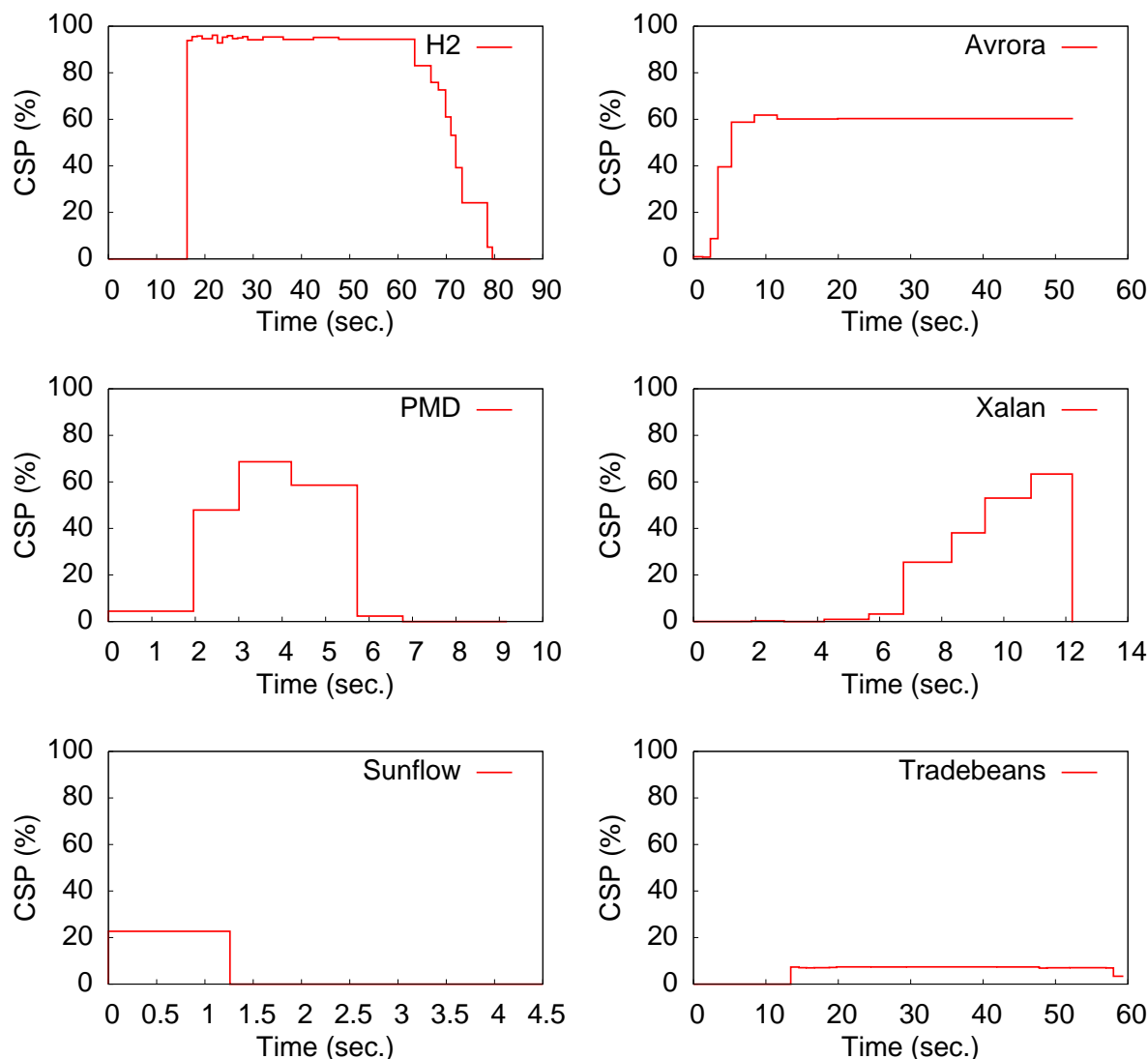
Figure A.3: CSP avec un intervalle de mesure d'une seconde.

d'étranglement soit complexe car les classes doivent impérativement être chargées séquentielle-ment.

Xalan est un parser XSLT transformant les documents XML en documents HTML. Le ver-rou associé avec l'objet `java.util.Hashtable` a une CSP moyenne de 20,4 %. `java.util.-Hashtable` utilise ce verrou pour assurer l'exclusion mutuelle sur chaque accès à la table de hachage, ce qui mène à un goulot d'étranglement. Comme présenté dans la Figure A.3, un seul thread remplit la table de hachage pendant la première phase (de 0 sec. à 6,8 sec.) et par con-séquent la CSP est négligeable. Cependant au cours de la deuxième phase (de 6,8 sec. à la fin), tous les threads de l'application accèdent la table de hachage, ce qui augmente la CSP jusqu'à 64,3 %. Cette CSP élevée est sous-estimée quand celle-ci est moyennée sur toute la durée de

l'exécution, ce qui rend son identification difficile sans la séparation en phases. Nous avons réimplémenté cette table de hachage en utilisant une `java.util.concurrent.ConcurrentHashMap` qui n'utilise pas de verrou global. Ce changement requiert la modification d'une seule ligne de code et améliore le temps d'exécution de l'application de 15 %. Cette analyse montre que cette information générée par Free Lunch peut en pratique aider les développeurs.

Sunflow est une application de rendu d'image. Le verrou associé avec l'objet `org.sunflow.-core.Geometry` a une CSP moyenne de 5,8 %. Comme présenté dans la Figure A.3, Sunflow affiche un pic de CSP modéré au début de son exécution. Cela apparaît pendant la tesselation des objets 3D qui est effectuée en exclusion mutuelle. Comme le nombre d'objets 3D est faible comparé au nombre de threads, de nombreux threads bloquent en attendant que la tesselation soit terminée. Afin d'améliorer les performances, il serait préférable de paralléliser le calcul de la tesselation.

Tradebeans simule un système de transactions d'actions en ligne qui inclut H2 pour stocker les données persistantes. Le verrou associé avec l'objet `org.h2.Database` a une CSP moyenne de 6,0 %. Ce verrou est également le goulot d'étranglement reporté dans l'application H2. Comme présenté dans la Figure A.3, une phase avec une CSP faible débute à 13,4 sec. et persiste jusqu'à ce que l'application se termine. Comme expliqué précédemment, de profondes modifications d'H2 seraient nécessaires afin d'améliorer les performances.

A.3.2.2   Cassandra

Cassandra [61] est une base de données distribuées NoSQL avec une architecture basée sur BigTable [16] et Dynamo [27]. Elle ne contient pas de point individuel de défaillance et est conçue pour passer à l'échelle; les données sont partitionnées et répliquées sur les nœuds. La durabilité est assurée par l'utilisation d'un journal des transactions validées qui enregistre toutes les modifications. Comme explorer ce journal pour répondre à une requête est coûteux, Cassandra maintient également un cache contenant l'état de la base de données. Ce cache est partiellement stocké sur disque et en mémoire. Après une panne, un nœud doit reconstruire ce cache avant de répondre aux requêtes des clients. Pour cela, il reconstruit le cache qui était stocké en mémoire en rejouant les transactions depuis le journal des transactions validées.

Un des développeurs de Cassandra a reporté un problème de performance dû à un verrou dans la version 1.0.0 de Cassandra[3]. Pendant cette phase, le temps de réponse était multiplié par 20. Ce problème a été observé sur une configuration où la base de données était déployée sur 3 nœuds avec un facteur de réplication de 3 et où la cohérence des données était assurée par un quorum avec 2 réplicas. Aucune autre information à propos de la configuration n'a été fournie. Par conséquent, nous avons été incapable de reproduire ce problème.

Bien que nous n'ayons pas été capable de reproduire ce problème, nous avons pu utiliser Free Lunch pour détecter une phase avec une CSP élevée dans Cassandra 1.0.0. En utilisant la configuration décrite précédemment, nous avons créé une base de donnée de 10 Go et utilisé le benchmark YCSB [77] pour stresser Cassandra avec un workload constitué de 50 % de lectures et de 50 % de mises-à-jour. Après 5 minutes 30 secondes, nous avons simulé une panne en stoppant

---

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

un nœud et en le redémarrant immédiatement après. Pendant la reprise, Free Lunch reporte une phase de CSP élevée d'environ 50 % avec un pic à 52 %. La phase de CSP élevée survient pendant le rejeu du journal des transactions validées et dure 11,4 sec.. Par chance, la section critique impliquée est la même que celle qui a causé le problème rapporté par le développeur dans la version 1.0.0 de Cassandra. En dehors de cette phase, la CSP pour ce verrou est proche de 0 %. La durée de cette phase de CSP élevée est proportionnelle à la taille du journal des transactions validées qui est elle-même proportionelle au nombre des modifications avant la panne. Ce résultat montre que Free Lunch est capable d'identifier précisément les variations de CSP pendant les phases dans des applications Java complexes. Cette phase est masqué par les autres profilers car les nœuds Cassandra ont généralement des temps d'exécutions de plusieurs jours.

Cette expérimentation illustre également la difficulté à produire et reproduire des problèmes de contention de verrous. En effet, ce scénario de test particulier est complexe à déployer et implique une panne de serveur, ce qui est relativement inhabituel. Pour cette raison, nous pensons que la probabilité de rencontrer ce problème pendant un test *in-vitro* est faible et que par conséquent, le profilage *in-vivo* est essentiel.

A.4   Conclusion

Cette thèse a présenté Free Lunch, un nouveau profiler de verrous conçu spécifiquement pour identifier les *phases* de contention élevées des verrous *in-vivo*. Le rôle de Free Lunch est d'identifier les verrous qui ralentissent le temps de réponse et le débit de l'application, en insistant sur les applications de type serveur qui utilisent généralement des plateformes multicœurs et pour lesquelles ces propriétés sont cruciales. Nous introduisons une nouvelle métrique appelée Critical Section Pressure (CSP) qui évalue le manque d'avancement des threads à cause de problèmes de synchronisation. La CSP est définie par le pourcentage de temps passé par les threads à acquérir un verrou par rapport au temps d'exécution cumulé de tous les threads. La métrique CSP est implémentée dans Free Lunch et aide à reporter aux développeurs le pourcentage de temps pendant lequel les threads sont bloqués et incapable de progresser à cause d'un verrou. Free Lunch est également conçu pour calculer régulièrement la métrique CSP afin de détecter les phases de contention des verrous qui pourraient survenir de manière imprévue à cause de divers facteurs externes difficile à reproduire *in-vitro*. Free Lunch est conçu de façon à limiter le surcoût d'exécution à ce qui est acceptable pour du profilage *in-vivo*. L'idée principale de l'implémentation est que Free Lunch est directement intégré à l'intérieur de la JVM, tirant parti d'un accès direct au sous-système de verrouillage et aux structures de données internes de façon efficace, et utilisant des primitives de gestion du temps optimisées. Cette approche est effectuée au détriment de la portabilité, cependant, Free Lunch est constitué de seulement 420 lignes de code ce qui devrait le rendre facile à implémenter dans une autre JVM

Free Lunch est évalué sur des applications provenant des suites de benchmarks de DaCapo 9.12, SPECjvm2008 et SPECjbb2005 et de la base de données Cassandra avec un workload de YCSB pour un total de 32 applications. Le matériel utilisé pour ces expérimentations est un serveur avec 4 processeurs AMD Opteron pour un total de 48 cœurs cadencés à une fréquence de

2,2 Ghz et 256 Go de mémoire vive. Le système tourne sous le noyau Linux version 3.2.0 64-bit provenant de la distribution Ubuntu 12.04. Nous avons identifié des phases de CSP élevée dans 6 applications. Certaines de ces phases sont masquées lors de l'utilisation de profilers existants, ce qui montre que Free Lunch est capable d'identifier des nouveaux types de goulot d'étranglement et de les reporter aux développeurs. Grâce à ces rapports, nous avons été capable d'améliorer les performances de l'application Xalan de 15 % en modifiant une seule ligne de code. Nous avons également trouvé une phase de CSP élevée dans la base de données Cassandra. Cette phase de CSP élevée survient pendant le rejeu du journal des transactions validées, effectué pendant la reprise après une panne d'un des nœuds afin de retourner dans un état cohérent. Nous avons étudié les métriques existantes des profilers de verrous faisant partie de l'état-de-l'art théoriquement et empiriquement sur des scénarios de synchronisation rencontrés typiquement dans des applications multithreadées. Nous avons trouvé que ces métriques ne renvoient pas de résultats pertinents pouvant aider les développeurs à détecter les verrous qui ralentissent le plus l'application. Nous avons évalué Free Lunch sur plus de 30 applications et montré qu'il ne dégrade jamais les performance de plus de 6 %. Ce résultat montre que Free Lunch peut être utilisé *in-vivo* pour détecter les phases où un verrou entrave l'avancement des threads dans des scénarios qui ne seraient pas nécessairement testés par un développeur *in-vitro*. Nous fournissons également une analyse détaillée des surcoûts d'exécution associés avec chaque choix de conception dans HPROF afin de comprendre l'origine de son surcoût d'exécution. Ensuite en partant de la version originale de HPROF, nous l'avons graduellement améliorée afin d'atteindre le design final de Free Lunch.

Bibliography

[1] A. Agarwal and M. Cherian. Adaptive Backoff Synchronization Techniques. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ISCA '89, pages 396–406, 1989.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, 2003.

[3] 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 Syst. J.*, 44(2):399–417, Jan. 2005.

[4] Alphaworks team Webpage. https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=18d10b14-e2c8-4780-bace-9af1fc463cc0, 2014.

[5] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance Analysis of Idle Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 739–753, 2010.

[6] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, 1967.

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

[8] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 258–268, 1998.

[9] S. Benedict, V. Petkov, and M. Gerndt. PERISCOPE: an online-based distributed performance analysis tool. In *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, pages 1–16, 2009.

[10] W. Binder. A Portable and Customizable Profiling Framework for Java Based on Bytecode Instruction Counting. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS '05, pages 178–194, 2005.

[11] W. Binder and J. Hulaas. A portable CPU-management framework for Java. *Internet Computing, IEEE*, 2004.

[12] 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 *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, 2006.

[13] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable Locks are Dangerous. In *Proceedings of the Linux Symposium*, July 2012.

[14] R. Bryant and J. Hawkes. Lockmeter: Highly-informative Instrumentation for Spin Locks in the Linux Kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference - Volume 4*, ALS '00, pages 271–282, 2000.

[15] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional Profiling for Multi-tier Applications. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, 2007.

[16] 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 *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 205–218, 2006.

[17] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, 2014.

[18] K. Coulomb, A. Degomme, M. Faverge, and F. Trahay. An open-source tool-chain for performance analysis. In *Parallel Tools Workshop*, pages 37–48. Springer, 2011.

[19] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang. Unicorn: A system for searching the social graph. *Proc. VLDB Endow.*, pages 1150–1161, Aug. 2013.

[20] C. Curtsinger and E. D. Berger. COZ: Finding Code that Counts with Causal Profiling. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, Work-in-progress, 2015.

[21] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA

Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394, 2013. ISBN 978-1-4503-1870-9.

[22] F. David. Profiler dynamique de contention pour les verrous des applications java. Com-PAS '13, 2013.

[23] F. David, G. Thomas, J. Lawall, and G. Muller. Continuously Measuring Critical Section Pressure with the Free Lunch Profiler. Research Report RR-8486, Inria Whisper, 2014. URL https://hal.inria.fr/hal-00957154.

[24] F. David, G. Thomas, J. Lawall, and G. Muller. Continuously Measuring Critical Section Pressure with the Free-Lunch Profiler. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 291–307, 2014.

[25] J. Davidson, B. Liebald, J. Liu, P. Nandy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston, and D. Sampath. The YouTube Video Recommendation System. In *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10, pages 293–296, 2010.

[26] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating System Design and Implementation*, OSDI '04, pages 107–113, 2004.

[27] 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 *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, 2007.

[28] J. Demme and S. Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 353–364, 2011.

[29] D. Dice. Implementing fast Java TM monitors with relaxed-locks. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM '01, pages 79–90, April 2001.

[30] D. Dice, M. Moir, and W. Scherer. Quickly reacquirable locks. Technical report, Sun Microsystems, 2003. URL http://home.comcast.net/~pjbishop/Dave/QRL-OpLocks-BiasedLocking.pdf.

[31] 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 *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 6 pp.–, April 2003.

[32] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle graphs: visualizing scalability bottlenecks in multi-threaded applications. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 355–372, 2013.

[33] Eclipse IDE.
https://eclipse.org/, 2014.

[34] M. S. M. B. et al. Détection automatique d'anomalies de performances. ComPAS '15, 2015.

[35] Extrae. http://www.bsc.es/computer-sciences/extrae, 2015.

[36] S. Eyerman and L. Eeckhout. Modeling Critical Sections in Amdahl's Law and Its Implications for Multicore Design. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 362–370, 2010.

[37] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. VMKit: A Substrate for Managed Runtime Environments. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 51–62, 2010.

[38] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 103–116, 2009.

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

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

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

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

[43] Y. Huang, Z. Cui, L. Chen, W. Zhang, Y. Bao, and M. Chen. HaLock: hardware-assisted lock contention detection in multithreaded applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 253–262, 2012.

[44] IBM Support Assistant.
http://www.ibm.com/software/support/isa, 2014.

[45] H. Inoue and T. Nakatani. How a Java VM can get more from a hardware performance monitor. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 137–154, 2009.

[46] Intel Trace Analyzer and Collector. https://software.intel.com/en-us/intel-trace-analyzer, 2015.

[47] Intellij IDEA. https://www.jetbrains.com/idea/, 2014.

[48] International Data Corporation. The Digital Universe of Opportunities. Technical report, International Data Corporation, April 2014. URL http://germany.emc.com/collateral/analyst-reports/idc-digital-universe-2014.pdf.

[49] Java Lock Analyzer. JLA homepage. http://ftparmy.com/125054-ibm-lock-analyzer-for-java.html, 2014.

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

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

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

[53] JVMDI. Java^{TM} Virtual Machine Debug Interface. https://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jvmdi-spec.html, 2014.

[54] jvmpi. Java^{TM} Virtual Machine Profiler Interface. http://docs.oracle.com/javase/1.4.2/docs/guide/jvmpi/jvmpi.html, 2014.

[55] JVMTI. Java^{TM} Virtual Machine Tool Interface. http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html, 2014.

[56] T. Kalibera, M. Mole, R. Jones, and J. Vitek. A Black-box Approach to Understanding Concurrency in DaCapo. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 335–354, 2012.

[57] K. Kawachiya, A. Koseki, and T. Onodera. Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 130–141, 2002.

[58] Y. Koren. Factorization meets the neighborhood: A multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 426–434, 2008.

[59] Y. Koren. Collaborative filtering with temporal dynamics. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 447–456, 2009.

[60] R. Lachaize, B. Lepers, and V. Quéma. MemProf: A Memory Profiler for NUMA Multicore Systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, 2012.

[61] A. Lakshman and P. Malik. Cassandra: A Structured Storage System on a P2P Network. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, 2009.

[62] D. Lea. The Java.Util.Concurrent Synchronizer Framework. pages 293–309, 2005.

[63] T. Liu and E. D. Berger. Sheriff: Precise detection and automatic mitigation of false sharing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, 2011.

[64] 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 *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, pages 65–76, 2012.

[65] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, 2005.

[66] Marissa Mayer at Web 2.0. http://glinden.blogspot.fr/2006/11/marissa-mayer-at-web-20.html, 2006.

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

[68] 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 '13, pages 849–866, 2013.

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

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

[71] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel. Developing scalable applications with vampir, vampirserver and vampirtrace. In *PARCO*, volume 15 of *Advances in Parallel Computing*, pages 637–644, 2007.

[72] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, 2012.

[73] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. *Supercomputer*, 12:69–80, 1996.

[74] T. Onodera and K. Kawachiya. A Study of Locking Objects with Bimodal Fields. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 223–237, 1999.

[75] T. Onodera, K. Kawachiya, and A. Koseki. Lock Reservation for Java Reconsidered. In *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 559–583, 2004.

[76] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, 2013.

[77] 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 *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 9:1–9:14, 2011.

[78] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 335–348, 2010.

[79] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, pages 17–31, mar 1995.

[80] F. Pizlo, D. Frampton, and A. L. Hosking. Fine-grained Adaptive Biased Locking. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 171–181, 2011.

[81] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: Black-box Performance Debugging for Wide-area Systems. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, 2006.

[82] K. Russell and D. Detlefs. Eliminating Synchronization-related Atomic Operations with Biased Locking and Bulk Rebiasing. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 263–272, 2006.

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

[84] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford. Open | speedshop: An open source infrastructure for parallel performance analysis. *Sci. Program.*, 16(2-3):105–121, Apr. 2008.

[85] S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006. ISSN 1094-3420.

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

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

[88] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 269–280, 2010.

[89] The OpenJDK Hotspot JVM. http://openjdk.java.net/groups/hotspot/, 2014.

[90] The Scalasca Peformance Toolset Architecture. http://scalasca.org, 2015.

[91] TIOBE index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, March 2015.

[92] F. Trahay, F. Rué, M. Faverge, Y. Ishikawa, R. Namyst, and J. Dongarra. Eztrace: A generic framework for performance analysis. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '11, 2011.

[93] F. Trahay, E. Brunet, M. M. Bouksiaa, and J. Liao. Selecting points of interest in traces using patterns of events. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '15, 2015.

[94] N. Vasudevan, K. S. Namjoshi, and S. A. Edwards. Simple and Fast Biased Locks. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 65–74, 2010.

[95] F. Xian, W. Srisa-an, and H. Jiang. Contention-aware scheduler: unlocking execution parallelism in multithreaded Java programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 163–180, 2008.

[96] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad Hoc Synchronization Considered Harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 1–8, 2010.

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

[98] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 221–234, 2005.

[99] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud '10, 2010.

[100] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 629–644, 2014.