POLITECNICO DI MILANO
Corso di Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Analysis of Lock/Unlock sequences in the DaCapo Benchmark Suite

Formal Languages and Compiler Group

Politecnico di Milano

Relatore: Prof. Giampaolo Agosta

Correlatore: Ing. Ettore Speziale

Tesi di Laurea di:

Gianola Daniele, matricola 750245

Anno Accademico 2010-2011

Contents

List of Figures

List of Listings

1 Introduction

The communication between processes is very frequent and it must be in a well-structured way, ensuring the determinism of operations.

Both in the case where two processes need a explicit communication and when they only want to share a portion of data, their interaction must be well synchronized and in the proper way.

In a Java system, as an example, a study reported that 19% of the total execution time was wasted by thread synchronization in an early version of virtual machine [7].

Even single-threaded applications may spend up to half their time performing useless synchronization due to the thread-safe nature.

In addition, the synchronization between processes requires a fair amount of computing power, and therefore an optimization which concerns it can be helpful.

Through this document we will analyze a Java based benchmark, with the aim of identifying peculiar characteristics to exploit, in order to make some optimizations on the execution time.

These characteristics should be frequently present in the analyzed code, so as to make effective all these optimizations.

To do this, we have found an ideal benchmark for our purpose, the *DaCapo Benchmark Suite*, written in Java and composed by different kind of programs.

It consists of a set of open source, real world applications with non-trivial memory loads [10].

This document is divided into two sections: in the first one we will discuss the most common techniques to manage synchronization between threads and in the second one we will list all the analysis performed on the benchmark.

The first common tecnique for granting correct Interprocess Communication acts by Disabling Interrupts, and it is based on disabling all interrupts just after entering a critical region and re-enabling them just before leaving it.

Then we will show the set of Wait-Free Synchronization techniques, which consist of ensuring that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes.

These tecniques mentioned above are not only focused on a Java environment, but they can be used for every thread-based system.

However, there are a lot of tecniques that ensure correct synchronization between Java threads, which mainly differ in speed.

Their name are Thin Locks and Ultra Fast Locking, and they are based on assumptions about the behavior of the thread and today they are the fastest techniques used for process synchronization in Java systems. We will list these algorithms before to proceed with the experimental analysis.

Our analysis will proceed with the instrumentation of the benchmark code, in order to trace all instances of lock-unlock operations in the benchmark.

On this porpuose we will create a *Profiling Tool* (section 3.2), which generates for every running benchmark a trace composed by lock-unlock sequences, and will build a tool to perform any locking algorithm running on the trace.

Then, with this trace of lock/unlock sequence, we will perform varius *Statistics and Analysis on the trace* (section 3.4), particularly regarding the chains of dominance in the sequences of the lock.

These analysis are important for our porpouses, with the aim of identifying features in the lock-unlock sequences that allow us to optimize a little part of running code.

Finally we will propose a version of an optimized algorithm that exploits these sequences of locks, speeding up the execution of a portion of code.

2 State of the art

Processes frequently need to communicate each other, sometimes for exchange resources or to synchronize each other. This communication must be in a well-structured way, ensuring the determinism of operations, because language-supported synchronization is a source of serius performance problems in many programs.

Even single-threaded applications may spend up to half their time performing useless synchronization due to the thread-safe nature [8].

As an example, in a Java environment it was reported that 19% of the total execution time was wasted by thread synchronization in an early version of virtual machine [7].

In this section are listed all the main techniques used to ensure proper synchronization between concurrent applications.

All these tecniques are focused on systems based on Java language, because the benchmark we have analyzed and the modules we have created within this work use Java language.

After mentioning the importance of correct Interprocess Communication, we will show techniques that permit correct process communication listed in increasing order of complexity, starting from Interrupts Disabling, which is based on disabling all interrupts just after entering a critical region and re-enabling them just before leaving it.

Then we move to Wait-Free Synchronization techniques, which consist of ensuring that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes.

Finally, we conclude with Thin Locks and Ultra Fast Locking. The first one, Thin Locks, are based on the following assumption "*in a running real program the majority of lock acquisistions are uncontended*".

The second one, Ultra Fast Locking are based on Thread Locality principle, which claims that the locking sequence of a lock contains a very long repetition of a specific thread.

2.1 Interprocess Communication

As mentioned above, Interprocess Communication is a critical part in any realtime application, and therefore it assumes great importance.

If two or more processes wants to share resources they must establish a well synchronized communication, in order to avoid deadlocks and starvations.

There are many algorithms and techniques that allows processes to communicate in a proper way, and there are two key concepts involved in processes communication: *Race conditions* (subsection 2.1.1) and *Critical Regions* (subsection 2.1.2).

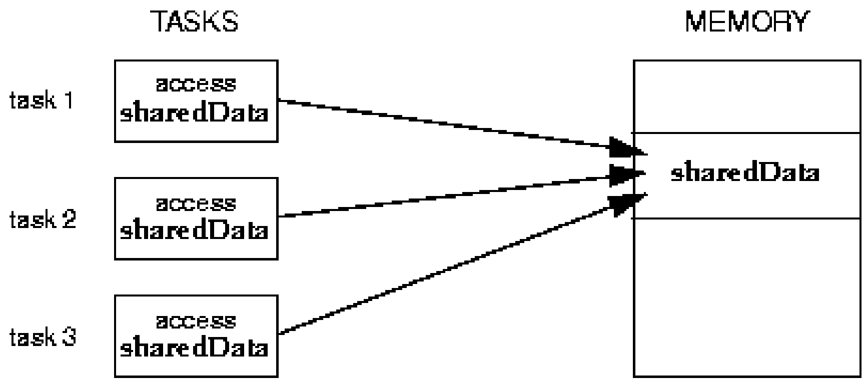
These concepts are not restricted to the sphere of Java applications, but they involve all the processes that need to communicate explicitly or only want to share some data.

Below we explain the basic information about these concepts, which are the key points for proper synchronization between processes.

2.1.1 Race Conditions

When two or more processes are working together, they may share some common data. So they have to read and write some shared resources, and if the final result depends on who runs precisely when, this situation is called Race Condition. An example of Race Condition in shown in figure 2.1, where 3 tasks are attemping to access the same shared data.

Figure 2.1: An example of Race Condition: three tasks are trying to access the sharedData at the same time.



2.1.2 Critical Regions

In order to avoid Race Conditions, is necessary to prohibit more than one process from reading and writing the shared data at the same time: this requisite is called *Mutual Exclusion*. In a lot of operating system, the most used solution for achieving Mutual Exclusion is by using primitive operations, and the part of the program where the shared data is accessed is called *Critical Region*. To avoid races, we must arrange matters such that no two processes were ever in their critical regions at the same time. Infact, this requirement is not sufficient for having parallel processes correctly and efficiently using shared data [21]. There are 4 conditions to hold:

1. Two processes can't be at the same time in their Critical Regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

These concepts, which may seem too theoretical, are actually the basis for proper management of synchronization between processes, and therefore must be guaranteed their correct implementation and optimization.

2.2 Disabling Interrupts

The most trivial way to avoid Race Conditions is by disabling the IRQ in a CPU.

This is the simplest solution and consists of disabling all interrupts just after entering a critical region and re-enabling them just before leaving it [21]. When interrupts are disabled, no clock interrupts can occur: in this way CPU can't switch between processes and the active process can access his shared data without fear.

This solution is working if and only if the system is with only one CPU, because in a multi-CPU scenario the other CPU will continue running and other processes can access the shared data.

Hence, this solution is useful only when the operating system has to access variables necessary for its functioning, but is not appropriate as a general mutual exclusion mechanism for user processes.

In the following section we will show more recent techniques used to optimize locking algorithms. Particularly, in sections 2.4 and 2.5, we will see both *lightweight locking* and *ultra fast locking*.

The first are focused on avoiding as much as possible the use of "heavy-weight" operating system mutexes and condition variables to implement Java monitors. The assumption behind these techniques is that most lock acquisistions in real programs are uncontended. Lightweight locking techniques use atomic operations upon monitor entry, and sometimes upon exit, to ensure correct synchronization, These techniques fall back to using OS mutexes and condition variables when contention occurs.

The second type of locking algorithms rely on the further property that not only are most monitors uncontended, they are only entered and exited by one thread during the lifetime of the monitor: this principle is called *Thread Locality* 2.5.1. Such monitors may be profitably biased toward the owning thread, allowing that thread to enter and exit the monitor without using atomic operations. If another thread attemps to enter a biased monitor, even if no contention occurs, a relatively expensive bias revocation operation must be performed. This optimization brings the benefit of the elimination of atomic operations being higher than the penalty of revocation.

2.3 Wait-Free Synchronization

Before taking a look in faster locking algorithms, we will list all the wait-free concurrent implementations.

A *wait-free* implementation of a concurrent data object is one that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes [15].

The wait-free condition provides fault-tolerance: no process can be prevented from completing an operation by undetected failures of other processes.

This feature is guaranteed by a series of operations at low level, which will be listed in the following sections.

Starting from Consensus Number, where each object has associated a maximum number of processes for concurrent accesses on it, then we will see Atomic Registers, which consists of atomic operations between two processes.

Then we procede with a set of operations which perform a series of low-level instructions, such as Read-Modify-Write Operations, Queue, Stacks, Lists and Memory-to-Memory Operations.

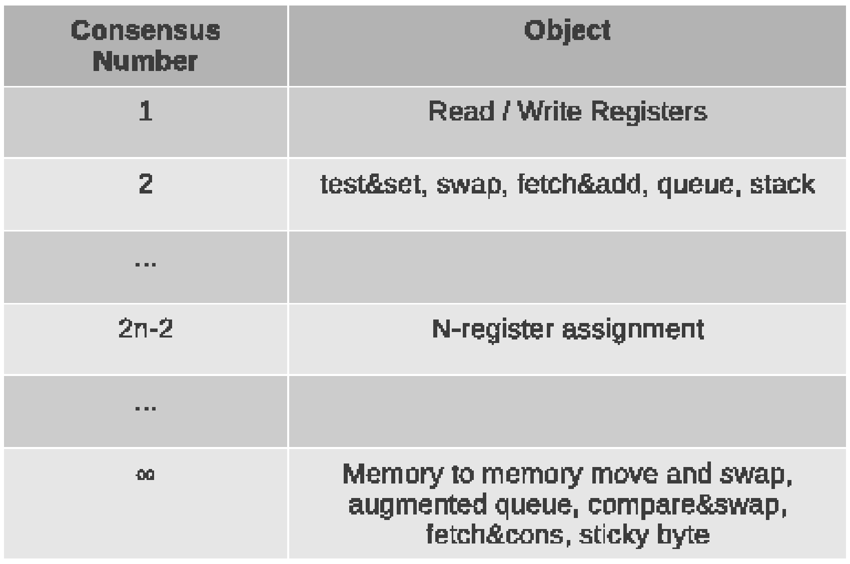
2.3.1 Consensus Number

The *Consensus Problem* is a common problem in distributed computing, it consists of achieve overall system reliability in the presence of a number of faulty processes. This often requires processes to agree on some data value that is needed during computation.

In [15] authors propose an hierarchy of objects (see fig. 2.2), such that no object at one level can implement any object at higher levels.

Each object has an associated *consensus number*, which is the maximum number of processes for concurrent processes, we show that it is impossible to construct a wait-free implementation of an object with consensus number n from an object with a lower consensus number.

Figure 2.2: Consensus hierarchy: each object has an associated consensus number, which is the maximum number of processes for concurrent processes.

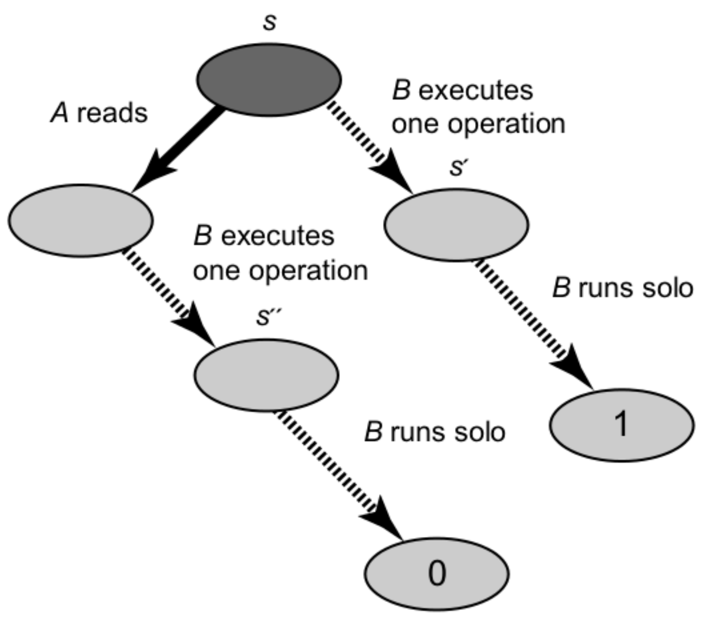


Consensus Number	Object
1	Read / Write Registers
2	test&set, swap, fetch&add, queue, stack
...	
2n-2	N-register assignment
...	
∞	Memory to memory move and swap, augmented queue, compare&swap, fetch&cons, sticky byte

2.3.2 Atomic Registers

With *atomic registers* is impossible to implement two-process consensus protocol, because this wait-free implementation ha consensus number 1. Suppose two threads, A and B, which want to perform a consensus protocol with atomic registers in different situations.

Figure 2.3: Case 1: A reads first, but B runs solo and eventually decides 0 or 1.



The results are shown in the following figures.

In fig. 2.3, if B moves first, driving the protocol to a state s', then B runs solo and eventually decides 1. If A moves first, B then runs solo starting in s" and eventually decides 0.

In figure 2.4 there are two possible execution scenarios.

If A writes first, the resulting protocol state is 0, B then runs solo and decides 0. If B writes first, the resulting protocol state is 1, so A runs solo and decides 1.

The problem is that B cannot tell the difference between s' and s", so B must decide the same value starting from either state, a contradiction [14].

In figure 2.5, A is about to write to r_0 and B to r_1.

If A writes to r_0 and the B writes to r_1, the resulting protocol result is 0, because A went first.

If B writes to r_1 and then A writes to r_0, so the resulting protocol state is 1 because B went first.

Figure 2.4: Case2: A and B write on the same register, if A writes first, the resulting protocol state is 0, B then runs solo and decides 0. If B writes first, the resulting protocol state is 1, so A runs solo and decides 1.

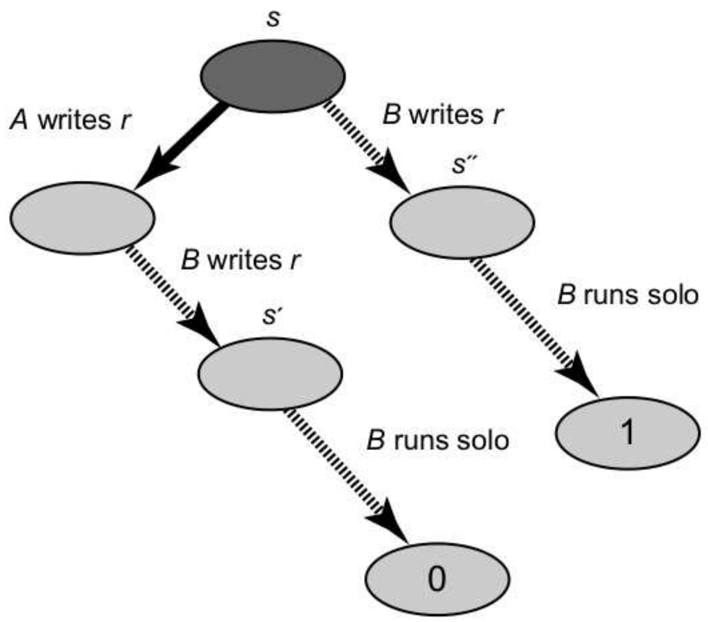
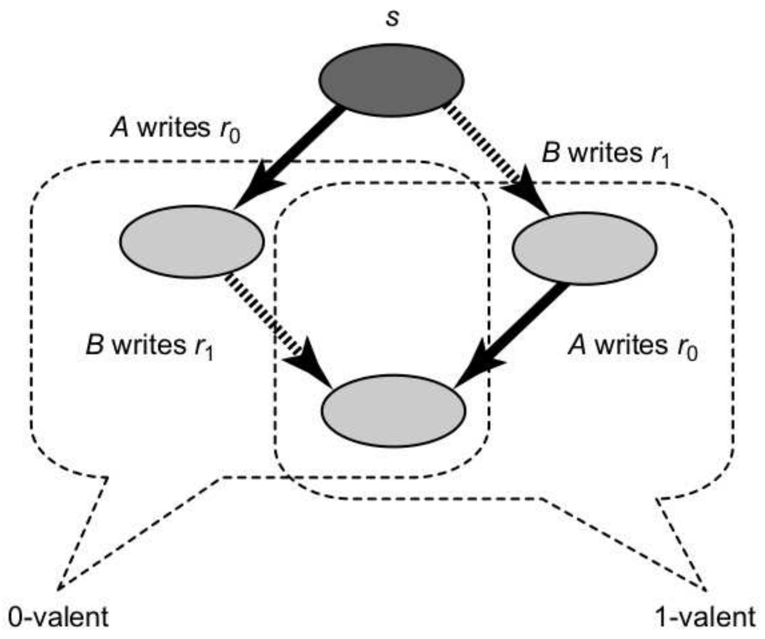


Figure 2.5: Case 3: A and B write on different registers. Here the result depends on who starts, if A went first the result is 0, otherwise B went first and the resulting state is 1.



2.3.3 Read-Modify-Write Operations

These operations, such as *test&set*, *swap*, *compare&swap* and *fetch&add*, have consensus number 2. So they are more powerful tha read/write registers, and they permit consensus protocol with two threads.

The *compare&swap* (see listing 2.1) is a primitive used in the following sections, so we will explain how it works.

This primitive takes two values: *old* and *new*. If the register's current value is equal to *old*, it is replaced by *new*; otherwise is left unchanged. The register's old value is returned.

Listing 2.1: Compare and Swap.

```
compare-and-swap(r: register, old: value, new: value)
  returns(value)
  previous := r

  if previous = old
    then r := new
  end if

  return previous
end compare-and-swap
```

2.3.4 Queue, Stacks and Lists

Even *queues*, *stacks* and *lists* have consensus number of 2. This is due the structure of the FIFO, with two operations: `equeue` and `dequeue`.

2.3.5 Multiple Assignment

Atomic m-registers have consensus number at least 2m-2, so they can perform a consensus protocol with 2m-2 threads [15].

2.3.6 Memory-to-Memory Operations

These are a collection of atomic read/write registers having one additional operation *move*, which automatically copies the value of one register to another. The consensus number of this class of operations is infinite, so they can solve a consensus problem for a set of infinite processes.

All the techniques mentioned in this section are useful for managing synchronization for a growing number of concurrent processes. Indeed with *test&set*, *swap*, *compare&swap* and *fetch&add* operations we can assure correct synchronization between two processes, while with Memory-to-Memory Operations we can perform a correct synchronization between an infinite number of processes.

2.4 Thin Locks

Thin Locks [8] is a Java algorithm, developed by IBM Research Center, that allows lock and unlock operations to be performed with only a few machine instructions in the most common case.

This algorithm follows the assumption *"in a running real program the majority of lock acquisistions are uncontended"* and optimizes the code execution by ensuring that the process that owns the object being able to access it very quickly without having to check synchronization issues with other processes.

After this we will see the differences between Fast Path and Slow Path, which are the paths for uncontended and contended lock acquisistion.

Then we'll see the characteristics of Thin Locks, their structure in memory and their implementation.

Finally, we will give an example of the application of this algorithm in two different contests, a Locking/UnLocking sequence in a Single Thread environment and in a Multi Thread enviroment.

2.4.1 Fast Path and Slow Path

The JVM has separate code paths for contended lock acquisistion, called *Slow Path*, and for the uncontended one, called *Fast Path*.

Contended means that more than one thread wants to access a shared mutable variable.

These locks are very important, because they are the key for a correct synchronization among threads.

When a lock is neeeded by a single thread, the lock acquisistion is called uncontended: the owner of the lock can access the object whenever he want.

Thin Lock is an optimization of this case of lock acquisitions, following the assumption *"in a running real program the majority of lock acquisistions are uncontended"*. Then, improving the performance of uncontended locking, this algorithm can improve overall application perfomances.

2.4.2 Characteristics of Thin Locks

A common way to prevent Race Conditions is the use of Monitors.

This solution consists of a language level construct for provinding mutually exclusive access to shared data structures in a multi-threaded enviroment.

To our disadvantage, the overhead required by the necessary locking has generally restricted their use to relatively "heavy-weight" objects.

Java uses monitor semantics, and the methods of an object may be declared `synchronized`, meaning that the object must be locked for the duration of the method's execution [8]. In Java common classes such as `Vector` and `Hashtable` have a lot of `synchronized` methods. Hence, when there is an absence of any true concurrency, we can see a substantial performance degradation.

The *Thin Lock* algorithm dedicates a portion of each object as a look, in order to speed up synchronization. In current Java implementations Monitor are kept outside of the objects to avoid the space costs, and are looked up in a monitor cache. This solution is inefficient and it doesn't scale, because the cache itself must be locked during lookups to prevent Race Conditions with concurrent modifiers. In addition, if large numbers of synchronized objects are created, the space overhead of the monitor structures may be considerable.

Thin Locks have been implemented to achieve following characteristics:

- **Speed** Without contention both initial locking and nested locking are very fast.
- **Compactness** Only 24bit in each object are used for locking, but object size is not increased due to other space compression techniques.

- **Scalability** It includes global locks and provides synchronization instructions broadcasted over the global bus, but all these features are kept to an absolute minimum, allowing efficient execution on large multiprocessors.
- **Simplicity** Thin Locks are implemented as a coverage over the existing heavy-weight locking facilities.
- **Maintainability** The code is fully portable, assuming only the existence of a `compare-and-swap` operation.

Hence, the goal reached by this algorithm is a very low overhead for single-threaded programs, but also with excellent performance in the presence of multithreading and contention.

2.4.3 The Implementation

Before analysing the structure of the Thin Lock algorithm, we will take a look to the most common cases in locking an object.

Orderer by descending frequency, and with each scenario about an order of magnitude less common than the one preceding it, the common cases are [8]:

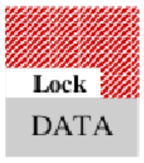
- Locking an unlocked object.
- Locking an object already locked by the current thread a small number of times
- Locking an object already locked by the current thread many number of times
- Attempting to lock an object already locked by another thread, for which no other threads are waiting.
- Attempting to lock an object already locked by another thread, for which other threads are already waiting.

Lot of benchmark analysis confirm that a median of 80% of all lock operations are on unlocked objects, and the nesting is very rare.

The enviroment where Thin Lock is implemented is assumed with a pre-existing heavy-weight system in place to support the full range Java synchronization semantics.

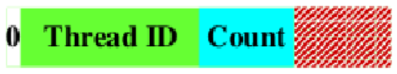
As far as hardware is concerned, there is an assumption which guarantees the portability and make everything more simple: we only assume the existence of a `compare-and-swap` operation, which is atomic. This operation takes three inputs: an *address*, an *old value* and a *new value*. If the contents of the address is equal to the old value, the new value is stored at the address ad the operation returns true. If the contents of the address is not equal to the old value, it returns false [8].

Figure 2.6: Object layout showing lock word, where the 24 bit reserved for the lock-word are obtained by using varius encoding techniques on the others values stored in the object.



The Thin Locks implementation reserves 24 bit for the lock-word (fig. 2.6), but not adding them to the objects header, simply obtaining them using varius encoding techniques for the others values stored in the object. The structure of the 24-bit lock allows the most common locking and unlocking operations to be performed with the minimum number of machine instructions. The lock field represents either a thin lock or a reference to a fat lock: the first bit value determines the type of the lock. If the object is not subject to contention, in other words has no needing of monitor semantics, the Thin Lock works very well.

Figure 2.7: Lock word structure for thin lock: the first bit is the shape bit, then there are 15 bits for the thread identifier and 8 for the nested deepth count.



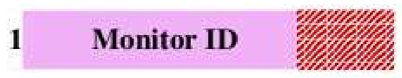
The structure of a Thin Lock is shown in figure 2.7, where the first bit is the shape bit. The following bits are shared out in this way: 15 for the thread identifier and 8 for the nested deepth count.

If the thread id is not zero, this value points to an entry in a system threads table.

These kind of objects are the majority, and with a not excessive nesting depth they are very efficient [8].

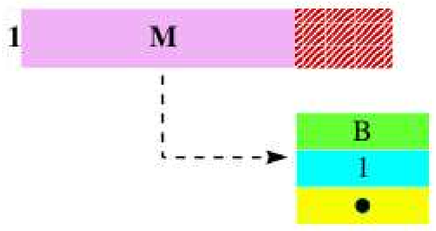
If the object is subject to inflated lock, the 24bit word structure remains the same but the values change their semantic (see fig. 2.8).

Figure 2.8: Lock word structure for inflated lock: the 24 bit word structure remains the same but the values change their semantic.



Particularly, while the shape bit remains the same but set to 1, the further 23 bits are used to point an entry in the Fat Lock table (see fig. 2.9). The Fat Lock contains the thread identifier and the count of lock counts, eventually the necessary queues and other fields.

Figure 2.9: Locked once by thread B: while the shape bit remains the same but set to 1, the further 23 bits are used to point an entry in the Fat Lock table.

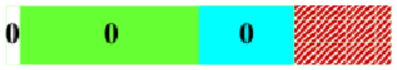


2.4.4 Locking/Unlocking with a Single Thread

The operation of locking with a single thread owning an object, in other terms without contention, starts with an unlocked object (see fig. 2.10).

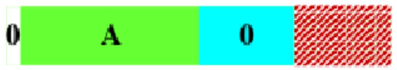
Then we assume that thread A wants to lock the object, so it performs a `compare-and-swap` operation on the word containing the lock field. The old value is

Figure 2.10: Unlocked object: all bits are set to 0 value



replaced and stored, for further uses, and as a result of this operation the new value is a shape equals to 0, the thread ID is set to ”A” and the count equals 0 (see fig. 2.11).

Figure 2.11: Locked once by thread A: the thread ID is set to *A* and the count equals 0.



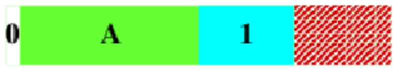
Count is mean as the number of nested lock minus one. When the thread A finishes its work with the object, it has to unlock it, then checks if it owns the object.

Instead of performing a `compare-and-swap`, it simply check if the value of the lock word is equal to the old value, and if so, store the new value in lock word.

If thread A once again locks the object, it will begin by performing the `compare-and-swap` operation, which will fail because the object is already locked by thread A itself.

The locking routine will check if the monitor shape is 0, that the thread index is equals to ”A” and the count is less than 255. If the check succedes, the count field is incremented by adding 256 to the lock word (see fig.2.11).

Figure 2.12: Locked twice by thread A: the thread ID is set to *A* and the count equals 1.



2.4.5 Locking/Unlocking with Multi Thread

Now we will explain how the algorithm works when two or more threads need an object.

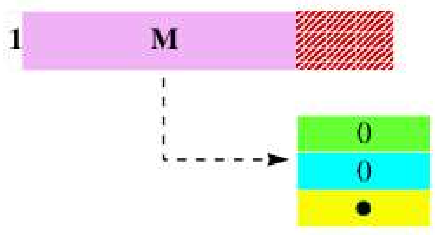
Assume that thread A has the object locked once, and thread B attemps to lock the object. Since the lock is owned by thread A and it is a Thin Lock, the thread B goes into a loop, called spin-locking loop.

When thread A releases the lock on the object, thread B who was "listening" on the object state, can obtain the lock. But now, instead of creating a Thin Lock on that object, thread B creates a Fat Lock on the object, assigning a monitor index to newly created monitor object, then change the shape bit with the value of 1. The result of this procedure is shown in figure 2.9.

Finally, when thread B unlocks the object, it remains in the inflated state, as shown in figure 2.13.

Up to now every lock on that object will use the fat lock, and if there is contention the fat lock discipline will handle the necessary queuing.

Figure 2.13: Unlocked object: the further 23 bits are used to point an empty entry in the Fat Lock table.



Thin Locks is a very fast solution that optimize Locking/Unlocking operations if the assumption on which it is based is true. If this assumption out to be false, this optimization is no longer possible and we must use a low-level and slower algorithm, or we must provide the system with a more sophisticated algorithm, such as the one listed in the following section.

2.5 Ultra Fast Locking

Here we will show the fastest way to ensure correct synchronization between two or more processes running in a Java environment.

This tecnique, called *Ultra Fast Locking* is unfortunately not always suitable into every contest, and in such cases a switch to a low lovel algorithm is performed, decreasing the performances of the entire application.

This algorithm is based on the *Thread Locality* principle, which claims that the locking sequence of a lock contains a very long repetition of a specific thread. The main way to exploit this principle is to reserve locks for thread, called Lock Reservation, which ensures a very fast acquisistion if the object is reserved for the thread.

Another similar technique, called Biased Locking, is based on eliminating atomic operations when the thread already owns the lock.

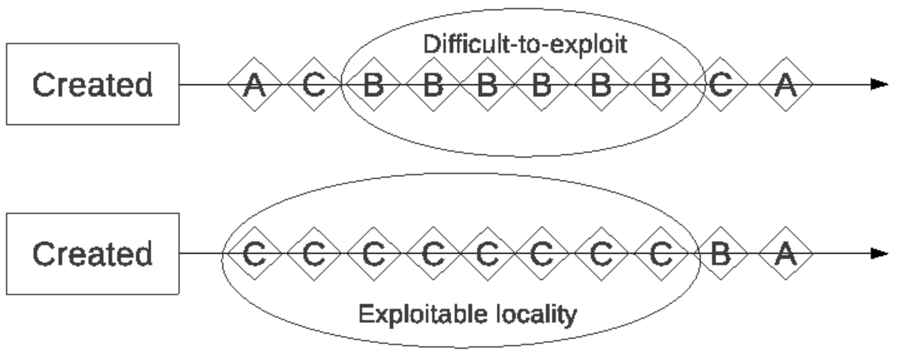
In the following sections we will see in the details all these techniques, showing example of locking scenarios and algorithms listings.

2.5.1 Thread Locality

In the following subsections we will see two important algorithms which optimize uncontended lock acquisitions.

They are based on the *Thread Locality* principle, which claims that the locking sequence of a lock contains a very long repetition of a specific thread.

Figure 2.14: Thread locality in two cases of lock sequences: in the second one is easy to exploit the sequence.



Thread locality of a lock is defined in terms of the *locking sequence*, the sequence of threads that acquire the lock [18]. The general form of thread locality is not easy to exploit, since we consider run time optimization and it is very hard to cheaply determine whether the lock exhibits thread locality or not.

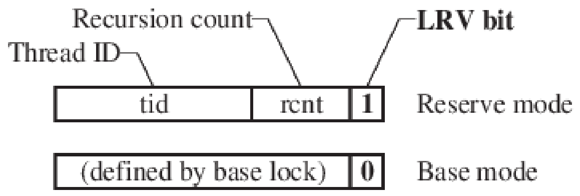
Nevertheless, there is a more easy way to determine if the lock sequence is dominated by a thread. For a given lock, if the locking starts with a very long repetition of a specific thread, this thread is the dominant locker and we can almost certainly say that we have thread locality [18].

2.5.2 Lock Reservation

The key idea for the Lock Reservation algorithm is to reserve locks for thread. When a thread attemps to acquire a lock, one of the following action is taken [18]:

- The object is reserved for the thread, so it can acquire the lock with few no-atomic instructions.
- The object is reserved for another thread, so the lock manager cancels the reservation, and runs a slower algorithm for further processing.
- If the object is not reserved, the lock manager uses a conventional algorithm.

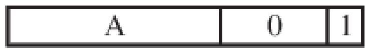
Figure 2.15: Lockword structure. A bit, called LVR, is used for representing the lock reservation status.



The algorithm reserves some space in the object header for a lock identifier, called *lockword* (see fig.2.15), where a bit is used for representing the lock reservation status.

This bit is called *LVR* and when is set the lockword is in the *reserve* mode, where Lock Reservation defines its own structure (fig. 2.15).

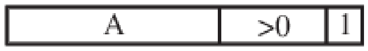
Figure 2.16: Lock Reserved for thread A, but not held by any thread.



In this case the lockword is in the reserve mode, and contains the owner thread identifier *tid* and the recursion count *rcnt*, which is the number of requests on the object performed by the thread.

If the count is zero, the lock is reserved but not held by any thread (see fig.2.16). If the count is non-zero, its value means the number of acquiring attemps perfomed by the thread (see fig.2.17).

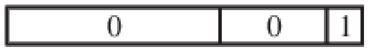
Figure 2.17: Lock Reserved and held by thread A



The recursion count is also used to control that a thread doesn't release a lock more times than it acquires the lock; this is intended *recursive locking*.

When the LVR bit is not set the lockword is in *base* mode and the structure is set by an underlying algorithm (fig. 2.15). When an object is created, the lock is *anonymously reserved*, and the lockword is in the reserve mode but not held by any thread (fig. 2.18).

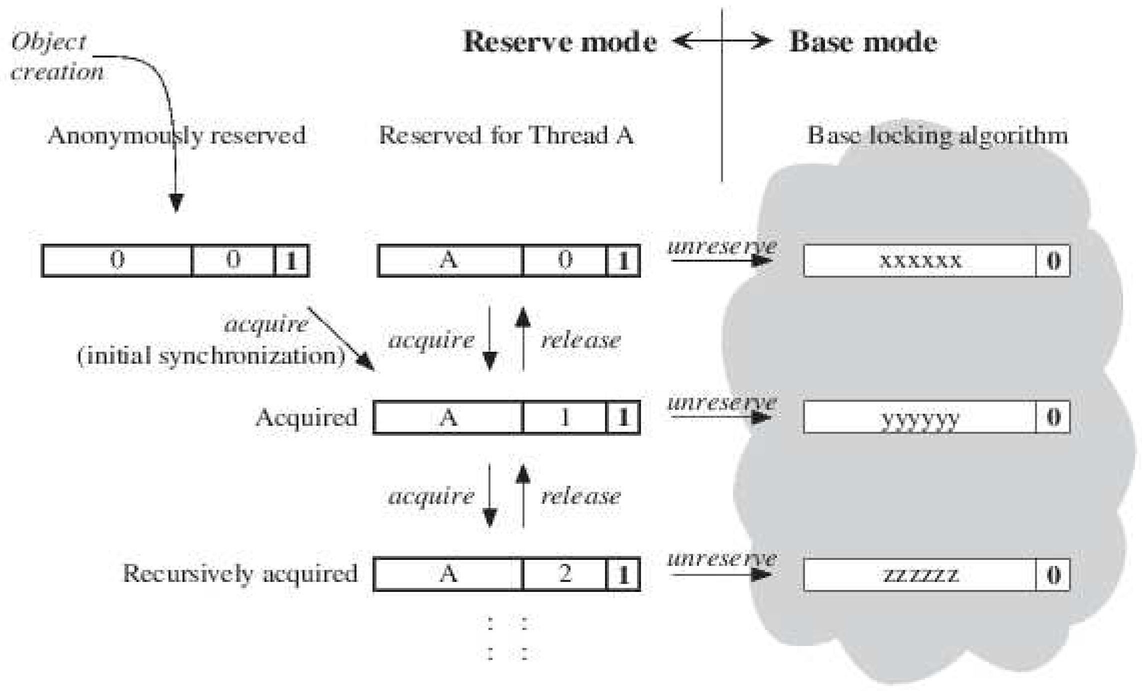
Figure 2.18: Reserved anonymously, the lockword is in the reserve mode but not held by any thread



When the reservation is canceled, the LVR bit is reset, and the lockword returns in base mode.

In figure 2.19) there is a full state transitions example, starting from the creation of the object, performing some acquisition-release by thread A and then switching from reserve mode to base mode more than once.

Figure 2.19: Lock state transitions: here there is an example of creation of the object, performing some acquisition-release by thread A and then switching from reserve mode to base mode more than once

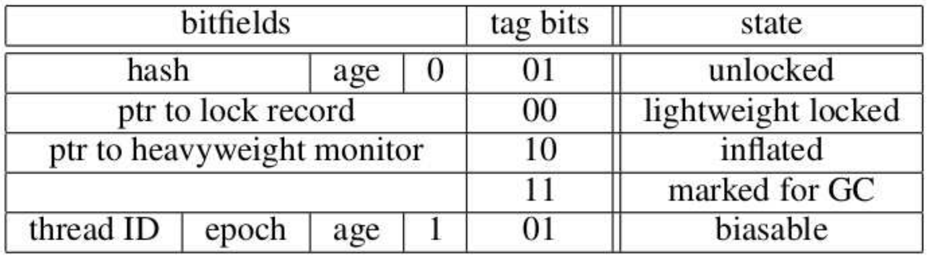


The very optimization and performance improvement (up to 53% in real Java programs [18]) is due to the fast acquisition of a lock when the thread has already reserved the object. In this case no atomic operations should be performed, thanks to the thread locality of Java locks.

2.5.3 Biased Locking

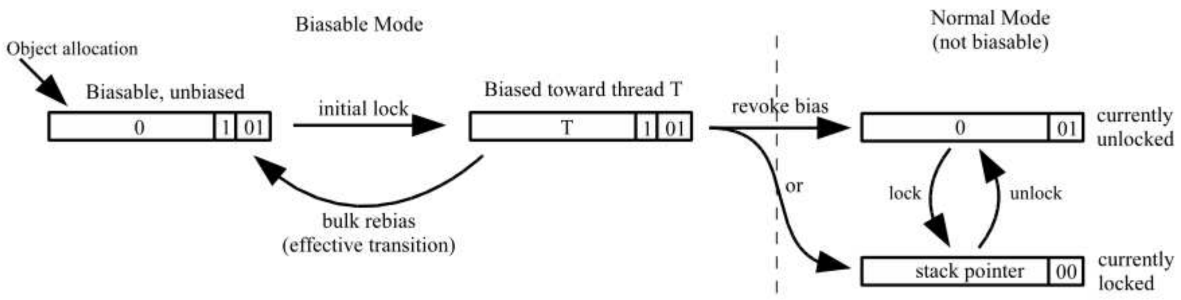
This Biased Locking algorithm, as the prevoius, is based on the thread latency exploit, in other words the property that monitors are entered and exited by one thread during the lifetime of the monitor. This tecnique, called SFBL (*Store Free Biased Locking*), is based on eliminating atomic operations when the thread already owns the lock, and it is very similar to the *lock revocation* tecnique [20].

Figure 2.20: Synchronization-related to be enabled of an object's mark word.



bitfields				tag bits	state
hash		age	0	01	unlocked
ptr to lock record				00	lightweight locked
ptr to heavyweight monitor				10	inflated
				11	marked for GC
thread ID	epoch	age	1	01	biasable

Figure 2.21: State transitions of an object's mark word under biased locking.



When an object is allocated and biasing is enabled for ites data type, a *bias pattern* is placed in the mark word indicating that object is *biasable*.

As shown in figure 2.20, the thread Id may be a pointer to the system internal representation of current thread or alternative schemes mat be used to allow better packing of thread IDs. In figure 2.20 we can see how Biased Locking works.

After object allocation, the object is biasable but unbiased: in order to obtain the biasable object, is performed an attempt with `compare-and-swap` instruction.

If the CAS succeeds, the object is now biasable for the current thread, which becomes the bias owner. If CAS fails, another thread is the bias owner, the thread bias is revoked and the object return in normal mode.

The bias owner can perform lock acquisitions without further work and no updates in the object header; this operation is very fast and it is based on thread locality.

The unlock operation perfoms no checks, the state of an object's mark word is tested to see if the bias pattern is still present.

If the check secceeds, the unlock operation finishes its work without any further tests.

This algorithm is almost similar to the Lock Reservation, but it can perform three further activities [20]:

- Bulk Rebiasing
- Bulk Revocation
- Epoch-based Bulk Rebiasing and Revocation

The *Bulk Rebiasing* is an optimization used to improve locking where biased locking is not profitable, such as producer-consumer queues where two or more threads are involved.

The goal is to disable the biased locking only for objects involved in continuos change of their owner. This tecnique adds, in an heuristic way, an effective cost to every data type, in order to estimate the cost of individual bias revocation.

When the cost exceeds a certain threshold, a bulk rebias operation is performed, by resetting the bias owner in the object header, for every objects of that data type.

If an object, for example involved in a producer-consumer program, needs further revocations, a *Bulk Revocation* is performed. In this case the mark words of all biasable instances of the object are reset to the initial value.

Particularly the algorithm works on data types, so the Bulk Revocation is performed on every objects of data type equal to that object.

The way of finding all instances of a certain data type is to walk through the object heap.

The *Epoch Bulk Revocation and Rebiasing* is performed by the use of a timestamp (see fig.2.20) called *epoch*, which indicates the validity of the bias.

An object is considered biased toward a thread T if both the bias owner in the mark word is T, and the epoch of the instance is equal to the epoch of the object's data type [20].

So each data type (class A for example) has a corresponding epoch as long as the data type is biasable. With epoch the rebiasing consists of incrementing the epoch number for the class A and scanning all thread stack to locate objects of class A that are currently locked, updating their bias epoch.

Objects whose epoch numbers were not changed currently, have a different epoch of their own class, so they are considered biasable but in unbiased state.

In listings 2.2 and 2.3 there are code example explaining both *epoch basec bulk rebiasing* and *epoch based bulk rebiasing and revocation*.

Listing 2.2: Biased locking acquisition supporting epoch based bulk rebiasing.

```
void lock(Object* obj, Thread* t)
{
  int lw = obj->lock_word;
  if (lock_state(lw) == Biased
      && bias_epoch(lw) == obj->class->bias_epoch)
  {
    if (lock_or_bias_owner(lw) == t->id)
    {
      // Current thread is the bias owner.
      return;
    }
    else
    {
      // Need to revoke the object's bias.
      revoke_bias(obj, t);
    }
  }
  else
  {
    // normal locking/unlocking protocol,
    // possibly with bias acquisition.
  }
}
```

Listing 2.3: Biased locking acquisition supporting epoch based bulk rebiasing and revocation.

```
void lock(Object* obj, Thread* t)
{
  int lw = obj->lock_word;

  if (lock_state(lw) == Biased
      && biasable(lw) == obj->class->biasable
      && bias_epoch(lw) == obj->class->
                           bias_epoch[obj->alloc_site_id])
  {
    ....

  }
```

3 Experimental analysis

In the following sections we will show the Benchmark we have chosen in order to perform some analysis on the running code.

This Benchmark, called *DaCapo Benchmark Suite*, is written in Java, and within a Java system we performed our analysis and we will propose a optimized locking algorithm.

Our analysis come with the hope of finding structures in the lock-unlock sequence, which allows us to make some optimizations on that sequence and then speedup the execution of the code.

On this porpuose we create a *Profiling Tool* (section 3.2), which generates for every running Benchmark a lock/unlock trace, used for the analysis and stored in a proper structure.

Then, with this trace of lock/unlock sequence, we performed varius *Statistics and Analysis on the trace* (section 3.4), particularly regarding the chains of dominance in the sequences of the lock.

Thus we created a sequence *Replayer* (section 3.3), which for now does a simple algorithm, but it can be used with more sophisticated algorithms.

With the data we generated we made some analysis on the sequence, particularly regarding Dominance Chains, Shared Objects Level and Hot Paths in the sequence.

We have also performed a search for particular sequences, which led us to hypothesize optimizations on the execution of the trace and then to propose an algorithm that exploits particularly lock-unlocking sequences and optimize that part of running code.

3.1 DaCapo Benchmark

In order to compute some statistics on lock-unlock sequence, we have chosen the DaCapo Benchmark Suite [12], a tool for Java Benchmarking by the programming language, memory management and computer architecture communities.

It consists of a set of open source, real world applications with non-trivial memory loads [10].

The choice was not causal, because this is the Benchmark for general purpose parallel applications for java. The version 9.12 that we used is the latest, released in 2009, which consists of 14 Benchmarks, listed in the following section.

3.1.1 List of Benchmarks

The 2009 release, consists of the following Benchmarks:

- **Avrora:** simulates a number of programs run on a grid of AVR microcontrollers
- **Batik:** produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik
- **Eclipse:** executes some of the (non-gui) jdt performance tests for the Eclipse IDE
- **Fop:** takes an XSL-FO file, parses it and formats it, generating a PDF file
- **H2:** executes a JDBC bench-like in-memory Benchmark, executing a number of transactions against a model of a banking application, replacing the hsqldb Benchmark
- **Jython:** inteprets a the pybench Python Benchmark
- **Luindex:** Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible
- **Lusearch:** Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible
- **Pmd:** analyzes a set of Java classes for a range of source code problems
- **Sunflow:** renders a set of images using ray tracing
- **Tomcat:** runs a set of queries against a Tomcat server retrieving and verifying the resulting webpages
- **TradeBeans:** runs the daytrader Benchmark via a Jave Beans to a GERONIMO backend with an in memory h2 as the underlying database

- **TradeSoap:** runs the daytrader Benchmark via a SOAP to a GERONIMO back-end with in memory h2 as the underlying database
- **Xalan:** transforms XML documents into HTML

For every Benchmark is possible passing as argument the *size* of the application, where the set of inputs is composed by *small*, *default* and *large*.

As the suite consists of real world applications, with a large variety of resources and memory accesses, DaCapo is the optimal candidate for our analysis.

Particularly, we needed a *trace* of the lock/unlock sequence, generated by running the DaCapo Benchmarks.

3.2 Profiling Tool

As mentioned above, this tool, written in Java, generates a lock/unlock sequence trace from running Benchmarks.

Listing 3.1: Synchronized example

```
int foo;

synchronized(foo)
{
  //do something with foo
}
```

Particularly, the tool acts in this way:

1. First, it inserts its own class-loader [17] in the place of the original one, modifying the DaCapo code.
2. Then, it loads every DaCapo Java class with this custom classloader.
3. Before loading the class, the tool looks for any entry of `MONITORENTER` and `MONITOREXIT` instructions in the class bytecode [16].

Listing 3.2: Synchronized method example

```
int c = 0;

public synchronized void increment()
{
  c++;
}
```

4. Using the Apache BCEL Library [9], it performs the instrumentation of every class by modifying every entry of `MONITORENTER` and `MONITOREXIT`, sorrounding them with a few tracing instructions.
5. Once the class is loaded and it is running, following the Observer Pattern [13] rules, a listener waits for every occurrence of lock/unlock events and reports them on apposite trace.

3.2.1 Notes

After the above, we must make few clarifications.

First, in a Java program the lock/unlock event doesn't occurs only at `MONITORENTER` and `MONITOREXIT` bytecode instructions. These low-level instructions are generated when a programmer uses the `synchronized` word to classify code-block. An example can be shown in the listing 3.1.

Hence, there is another way to generate a lock/unclok sequence, and this way is by using the word `synchronized` to classify a method, as the example in listing 3.2 can explain.

With these methods we could have traced lock/unclock events modifying the method bytecode, by adding tracing instructions before the method's first instruction and just before every `return` call.

Unfortunately, we had to face problems with this kind of methods, because often they didn't generate a well-formed trace.

A well-formed lock/unlock trace means a trace which can be parsed as a Dyck Language [19], but after several attempts we realized that the trace wasn't always well-formed.

Listing 3.3: Wait sorrounded by tracing instructions

```
public synchronized foo()
{
  TraceMonitorEnter(this);

  ...
  ...

  try
  {
    TraceMonitorExit(this);

    wait();
  }
  catch(Exception e)
  {
    if(!(e instanceof IllegalMonitorStateException))
      TraceMonitorEnter(this);

    throw e;
  }

  ...
  ...

  TraceMonitorExit(this);
}
```

Indeed sifting through source code, we have found `wait()` calls and `throw` of Exceptions inside `synchronized` methods .

These constructs could create problems with the trace, because if they occur the object is unlocked by the system and not by returning from the `synchronized` method, the one who saves the unlock event.

Listing 3.4: Throw sorrounded by tracing instructions

```
public synchronized boo()
{
   TraceMonitorEnter(this);

   ...
   ...

   TraceMonitorExit(this);
   throw e;

   ...
   ...

   TraceMonitorExit(this);
}
```

We fixed this problem by sorrounding every `wait()` with the code in listing 3.3 and every `throw` with a tracing istruction as showed in listing 3.4.

The `throw` case, the easiest, had been considered as a `return` from a method, because a throw of a new Exception behaves as an exit from a method.

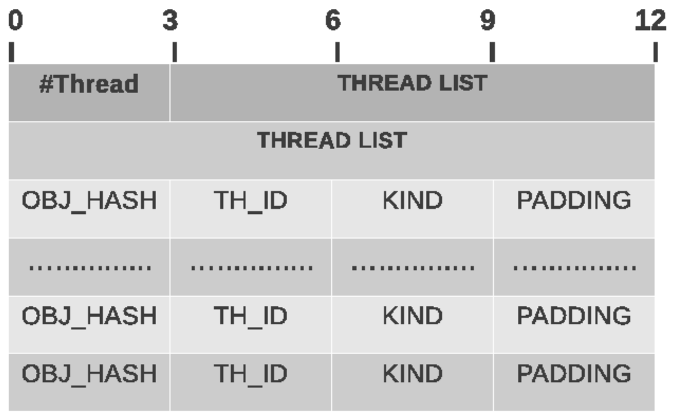
The more complicated case, the `wait()` call, can generate different behaves. As exaplained in [3], this call can generate different types of exceptions, for this reason it should be treated with a `try-catch` block.

A second clarification is about the trace format. We decided to create a trace file for every Benchmark execution, which structure is shown in figure 3.1.

In the figure 3.1, we can see how the trace is organized.

Particularly, the file is 12bytes aligned, according with 32bit integer values and zero-fill padding when the value is not integer. The trace consists of an header, with the

Figure 3.1: The structure of the trace: 12byte aligned trace, starting with the thread list.



information about the threads runned, followed by all the entries of every lock/unlock event.

In details:

- **Thread:** this field contains the number of threads used by the Benchmark
- **THREAD LIST:** here we stored every ID of every thread used by the Benchmark. Zero-fill padding is used to reach the 12byte alignment.
- **OBJ HASH:** is the hash of the object locked/unlocked by this event-entry.
- **TH ID:** is the ID of the thread which owns this entry.
- **KIND:** contains the information about the nature of the event, so if it is a lock or an unlock event.
- **PADDING:** Zero-fill padding.

The last clarification is about which Benchmark we took into account.

The DaCapo suite offers 14 Benchmarks, but we have skimmed them, because some did not generate trace, others used their own classloader and it was impossible to instrument their classes.

A single trace generated in a single execution with the chosen set of Benchmarks is about 650MB and it contains almost 50 millions entries. Thus, this simplification is not casual but is justified by the big amount of data we made with few Benchmarks.

3.3 Trace Replayer

Since the trace is like a Dyck language, where the balanced parentheses are represented by lock and unlock events, we have been able to create a parser for the trace.

This parser, written in C++, in addition to check if the trace is well formed, creates a tree representation of the events list.

When a thread needs to be launched, in the tree we have a bifurcation and two threads can run independently but competing on the same resources. This tree, showed in figure 3.2, is used by the trace replayer.

The replayer consists of some classes which implements the PThread Library [4] [6] and uses Strategy Pattern [13] (figure 3.3) to permit any kind of locking algorithm using that interface.

By implementing an interface composed by 2 virtual method called `lock()` and `unlock()`, ones can execute any lock algorithm.

The current implementation consist of a very simple algorithm, which uses PThread Mutex [4] [6] functions to lock and unlock resources.

3.4 Trace Analysis

After parsing the trace, we made a graph who represented the sequence of lock/unlock using the *.dot* file format [2] and *Graphviz* utility for showing it.

In figure 3.4 we can see an example of graph created from a trace; in this example the original trace is showed in listing 3.5, where *L(hash)* and *U(hash)* are respectively the representation of Lock and Unlock events, while *hash* is the Hash-Value of the locked object.

By this graph we made some analysis on the trace, as calculating the depth of the Dominance Chains of nested lock events, which show us how a thread behaves when it locks objects.

Then we found if that chain was Valid, which that means it reflects well-defined sequences of acquisitions in a program listing.

Figure 3.2: Threads representation with a tree. Every node is a Lock/Unlock operation on an object.

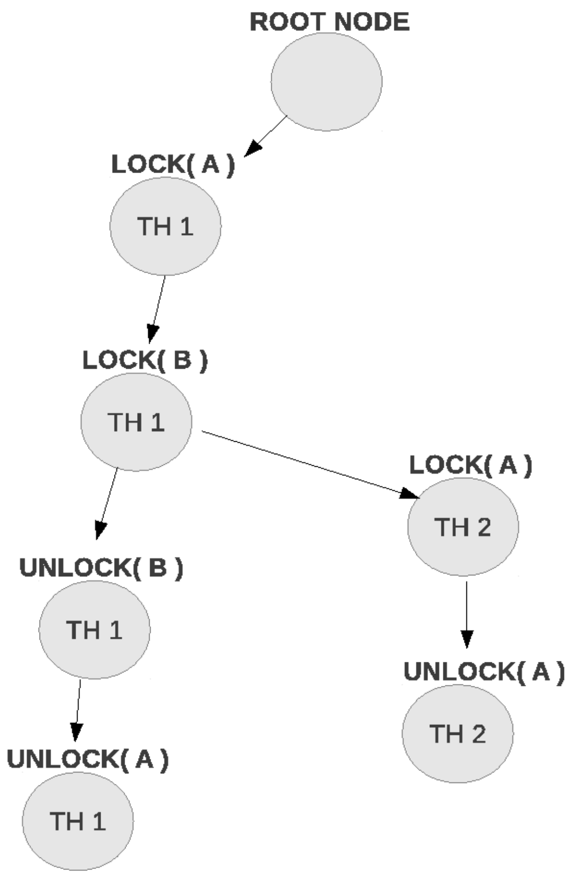


Figure 3.3: Strategy Pattern: it defines a family of algorithms, encapsulates each one, and makes them interchangeable. Then lets the algorithm vary independently from clients that use it.

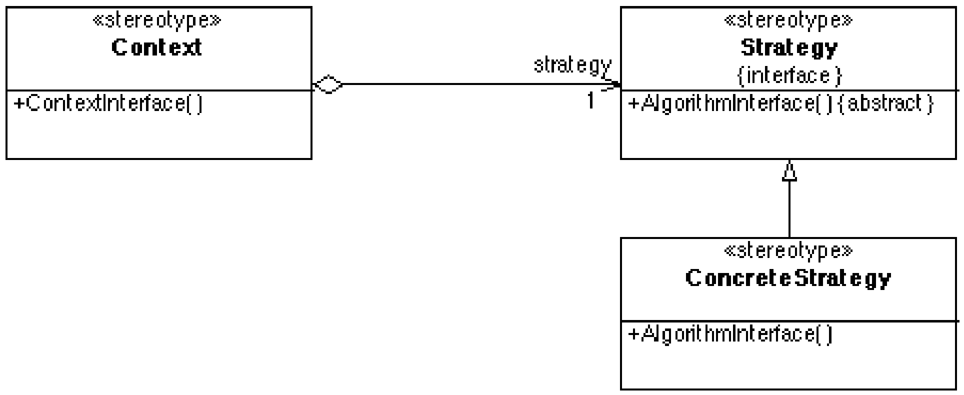
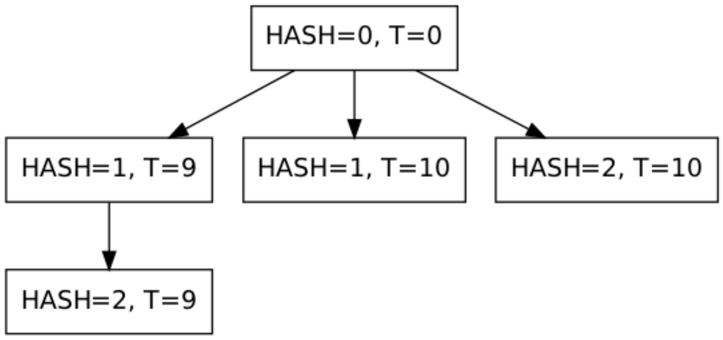


Figure 3.4: An example of trace graph: every branch represents a nested sequence owned by a single thread.



Listing 3.5: A simple trace example

```
Thread(2)  -> L(A)
Thread(2)  -> L(B)
                              Thread(1)  -> L(A)
                              Thread(1)  -> U(A)
Thread(2)  -> U(B)
Thread(2)  -> U(A)
                              Thread(1)  -> L(B)
                              Thread(1)  -> U(B)
```

So, with the sequence graph, in the following sections we will show the presence of cycles on the same object and we will give a weight to each object, by finding the level of sharing of each of them.

Finally, by combining all these analysis, we will show the presence of very crossed paths, calling them Hot Paths and relying on them to hypothesize a new algorithm.

3.4.1 Dominance Chains Depth

The main analysis performed on the trace was calculating the depth of the *Dominance Chains* of nested lock events.

Based on the graph mentioned above, we made a little change on a known graph algorithm: *Dijkstra's algorithm*, showed in listing 3.6.

This algorithm solves the single-source shortest path problem for a graph with non-negative edge path costs, producing the shortest path tree [11].

Our scenario was slightly different, every edge had the same cost and so we didn't have to look for the node with minimum distance, but it was enough to proceed with order in children. For this reason the algorithm used to create the shortest-paths tree has become essentially a *Breadth-first search* [11] algorithm.

In addition, since we were calculating a lower bound, we also needed an estimate of the reliability of the chains length. In order to obtain this value, in this algorithm we calculated the number of chains that had been deleted for each leaf node.

The resulting algorithm is shown in listing 3.7, and it is based on a Breadth-first search algorithm.

Listing 3.6: The Dijkstra Algorithm

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:
        dist[v] := infinity ;
        previous[v] := undefined ;
    end for ;
    dist[source] := 0 ;
    Q := the set of all nodes in Graph ;

    while Q is not empty:
        u := vertex in Q with smallest distance in dist[] ;
        if dist[u] = infinity:
            break ;
        end if ;
        remove u from Q ;
        for each neighbor v of u:
            alt := dist[u] + dist_between(u, v) ;
            if alt < dist[v]:
                dist[v] := alt ;
                previous[v] := u ;
                decrease-key v in Q;
            end if ;
        end for ;
    end while ;
    return dist[] ;
end Dijkstra.
```

Listing 3.7: The Algorithm that calculates the Lower Bound of Chains Length

```
//Initializing the FIFO with the root node
queue = [RootNode]

//Set of shortest paths tree
S = [R]

//Set of validNodes
validNode = []

while queue != empty:
    u = queue.pop()
    visited[u] = true

    //Adj[u] = set of u's children
    for v in Adj[u]:
        if visited[v] = false:
            queue.append(v)
            S.append(v)
            validNode[v] = true
        else
            //node already visited
            //here there is a compressed chain
            compressChains++
            validNode[v] = false
```

Figure 3.5: Lower Bound of Chains Length and the error committed in their evaluation.

Chain Depth	Number of chains	Compressed	% Error
1	684	490	72%
2	236	7	3%
3	2	0	0%

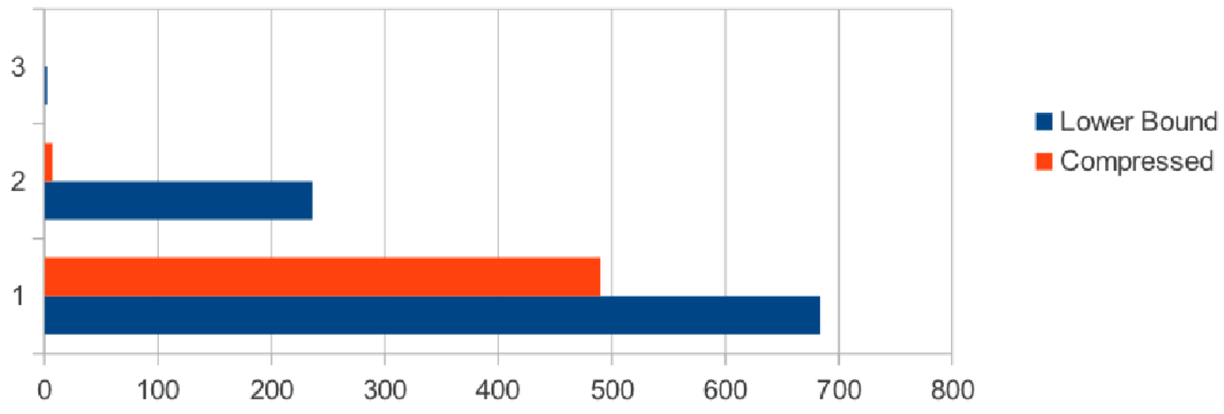
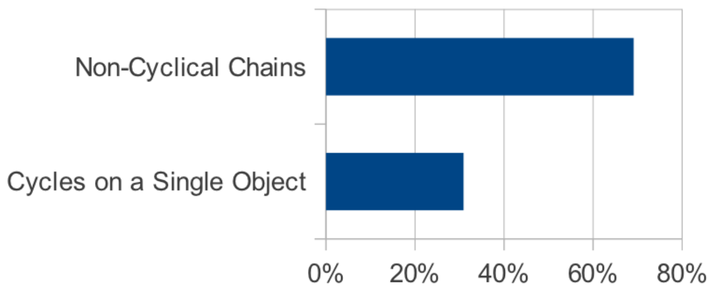


Figure 3.6: Percentage of nested chains on the same object. These cycles don't represent deadlock, they are owned by the same thread.

	Number of chains	% of chains
Cycles on a Single Object	271	31%
Non-Cyclical Chains	606	69%



In figure 3.5 we can see the resulting histogram, with the lower bound of chains length and the reliability of that value, calculated by analyzing the generated traces.

From the graph of the lower bound is possible to do some considerations about chains length, where the majority has a length less than 2:

Listing 3.8: Nested Synchronized Blocks are not very frequent

```
int foo, boo;

synchronized(foo)
{
 synchronized(boo)
 {

 }
}
```

- The majority of short chains means that a fast algorithm which has kept track of some chains that are often repeated and optimizes the sequence of lock acquisitions generated by them can not always be used.
- We must not forget that this is only a lower bound of chains length, in fact looking at the graph generated from the trace (see 3.4) you can find several chains of length greater than or equal to 2.

3.4.2 Cycles

With the algorithm mentioned above, we could calculate the eventual presence of cycles in the graph of events.

Due to the thread-based format of the trace, the presence of cycles is a warning for an event that is never welcome: the *deadlock* [1].

Fortunately, these cycles were only rings on the same node, which means that there were chains of dominance, where a thread attempts to acquire the same node in series.

This kind of lock acquisition, which is called *Recursive Lock* [5] and where the same thread can acquire the lock multiple times , it is a common mechanisms in Java native synchronization, where has been used since Java's inception in 1997.

In fig. 3.6 we can see that the percentage of chains where the lock sequence is on the same object is not relevant but not to overlook.

These chains are all of length greater than or equal to two, but since we can only estimate with certainty the length of the lower bound, we didn't take them into account in the following section.

3.4.3 Valid Chains

Listing 3.9: Depth-first postorder algorithm that looks for valid chains

```
//the list of valid chains
ChainsList = []

bool ValidChains(G, depth)
   for v in Adj[G]
      validSon = ValidChains(v,depth+1)
      //creates a new chain
      if depth == 0 or !validSon
         ChainList.newChain()

   if validNode[G]
      ChainList.append(G)
      return true

   return false
```

After calculating a lower bound of chains length, we tried to detect which chains can be called *valid*.

A valid chain reflects well-defined sequences of acquisitions in a program listing, which are repeated several times in the execution of that program.

Every object, in addition, has has an associated count of the number of references within the track, the *Reference Counter*.

Figure 3.7: An example of valid chain, which is composed by nodes A, B and C

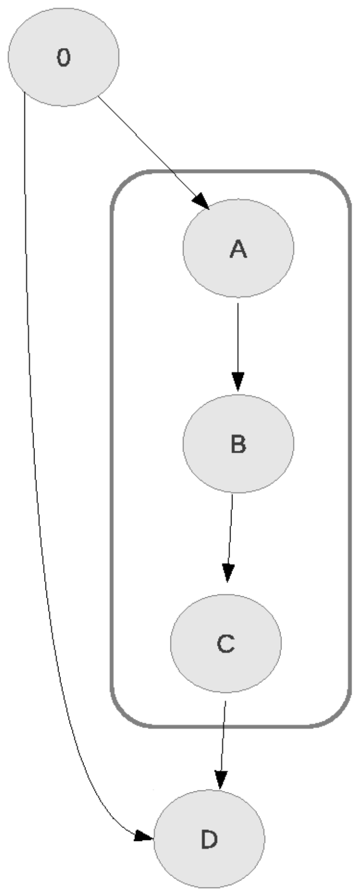
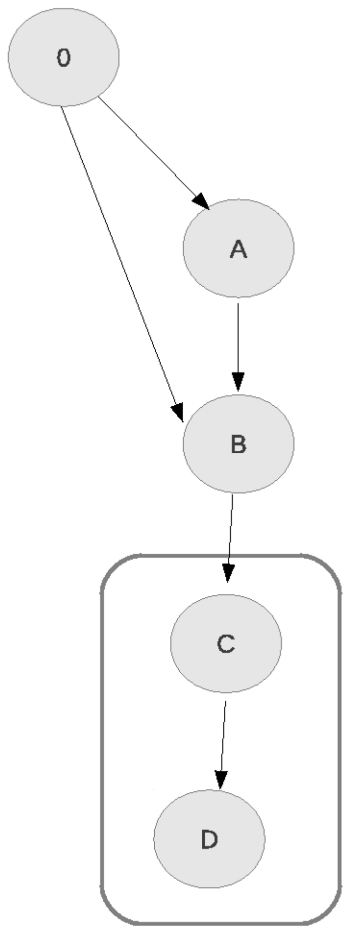


Figure 3.8: An example of valid chain, which is composed by nodes C and D



Combining this counter with chains, each chain can be associated with a probability that during the execution, after the acquisition of its first element, the second is really the next and so on, for its entire length.

This list of valid chains can be used to create an optimized algorithm which knows all of them and then can make very fast acquisitions assuming to know the objects that are part of the chain.

In figure 3.7 and 3.8 we can see two example of valid chains, and their main characteristic: each node must have only one father, so we are sure that the path of acquisition is unique.

In order to find these chains, we created an algorithm (see listing 3.9) which is based on the shortest-paths tree mentioned above. This algorithm is a *Depth-first search* algorithm [11], with a postorder evaluation of every node.

3.4.4 Shared Objects

In order to give a weight to each object, we find the level of sharing of each of them.

After creating the graph that connects each object to the thread that use it, we colored with warm colors the nodes that represent objects shared by many threads, and with cold colors the nodes that represent objects with a low degree of sharing.

In figure 3.9 is represented a portion of this graph, where one can see the difference there is between red nodes, which have many incoming arrows, and blue nodes, that have few incoming arrows.

The closer a node is shared, the more it can be called *hot* and more is its importance in a chain.

3.4.5 Maximum Spanning Tree

The label associated to each edge in figure 3.9 is the number of times that edge occurs in the trace.

An edge between two nodes A and B means that exists a sequence of lock formed by node A followed by node B. The number associated to this edge is the number of times that locking sequence occurs in the trace.

This number is another weight associated to a sequence, and in combination with the sharing level of each object we can find the most important sequences in the trace.

These sequences, called *paths*, are composed by nodes with high level of sharing and edges with an high number of reference.

Figure 3.9: Portion of Shared Nodes Graph: a cold color means low level of sharing, a warm color means an high level of sharing and white color represent a thread root node.

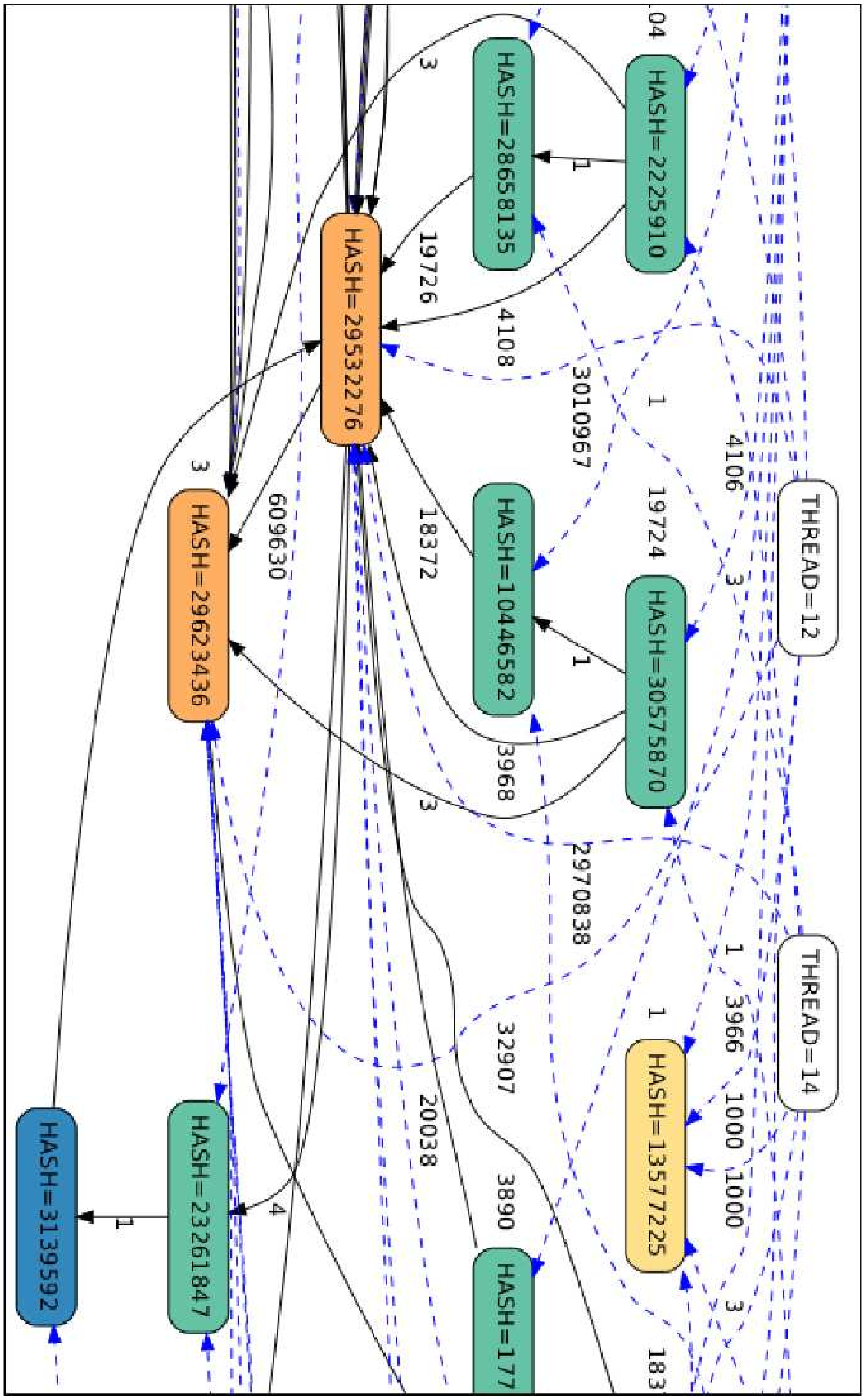
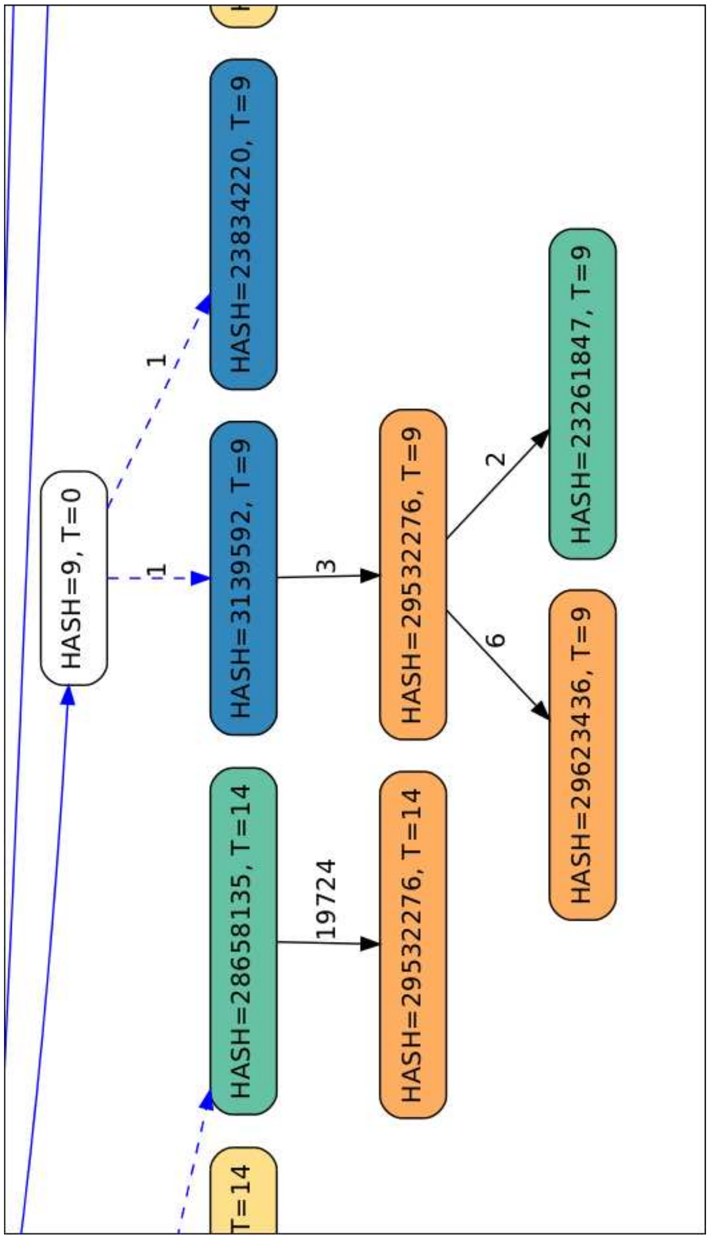


Figure 3.10: Portion of Maximum Spanning Tree



By an algorithm similar to the one used above (see 3.6), we have found the *Maximum Spanning Tree* of the trace and a portion of this tree is shown in figure 3.10.

These paths are very important for our goal, the feasibility analysis of a optimized locking algorithm, showed in section 3.6.

In the following section we filter these paths by giving weight to each node, by using the sharing level of each of them.

3.4.6 Hot Paths Tree

In the previous section we calculated the Maximum Spanning Tree by using the reference counter associated to each edge and so to each locking sequence.

Here we give weight to each node, by using the sharing level of each of them, and so we extract from previuos calculated sequences paths that really matter, composed of highly repeated sequences and very *hot* nodes.

The result (the H2's graph for example) is showed in figure 3.11, and is very simple to note the high importance of the sequence *29532276* followed by *29623436*.

These paths are called *Hot Paths*, and in following section we first show a detailed analysis of each Benchmark and at last we propose a locking algorithm that exploits the presence of these *Hot Paths* to optimize the execution of a Java program.

3.5 Benchmark Analysis

In this section we will list all the data obtained by previous analysis, by showing in detail all the important information regarding each Benchmark.

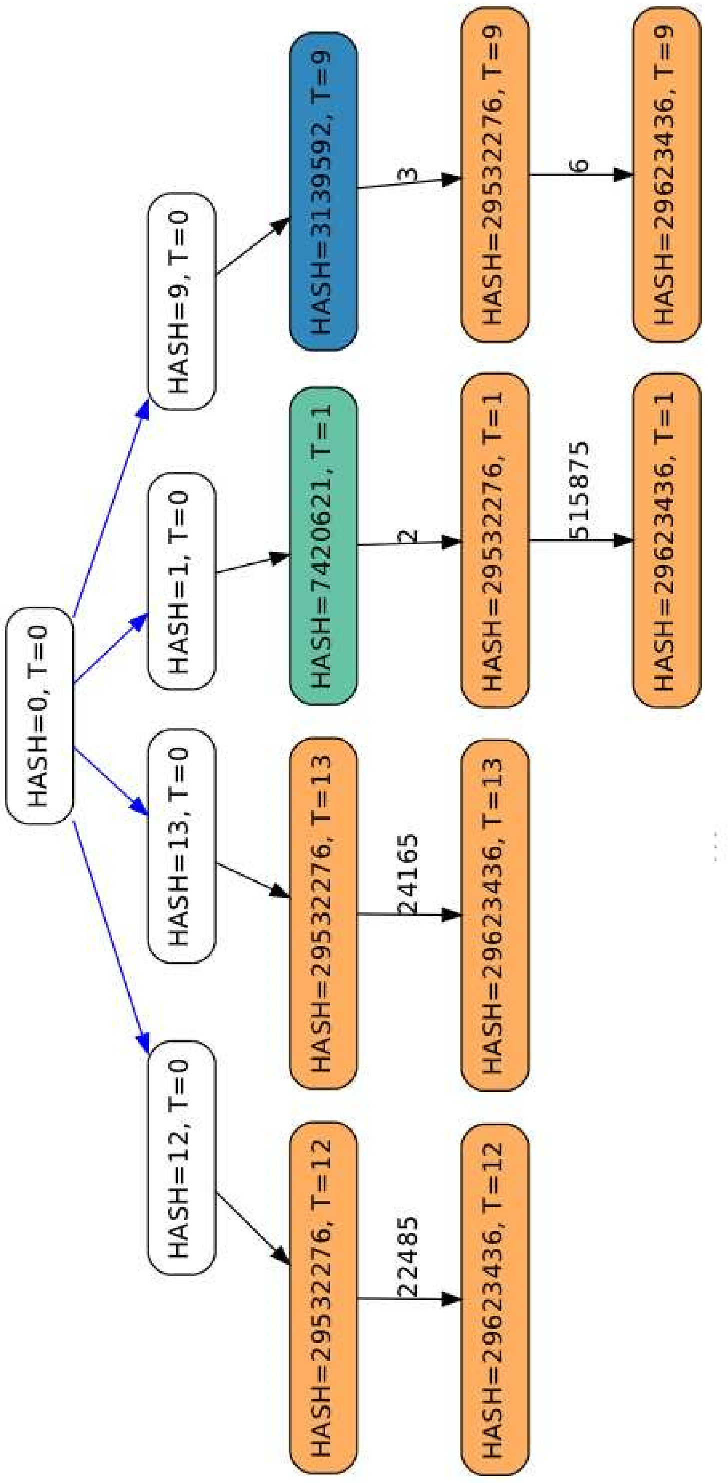
Not all the benchmarks led us to interesting meaning, but the one with more interesting informations covers the 90% of the trace, and justifies the lack of importance of the others.

As we will see, only the H2 Benchmark has a interesting behavior, but it consist of about 90% of all the trace and we won't commit errors omitting the others, by analyzing the algorithm we want to achieve.

Below we will list the results of our analysis, and particularly for each Benchmark we will show the sampling distribution of the Sharing Level, the Chains Depth and the presence of Hot Paths.

Then we will make some assumption concerning the applicability or not of an optimized algorithm, which exploits the presence of Hot Paths.

Figure 3.11: Hot Path Tree of H2 Benchmark



3.5.1 H2 Benchmark

Description

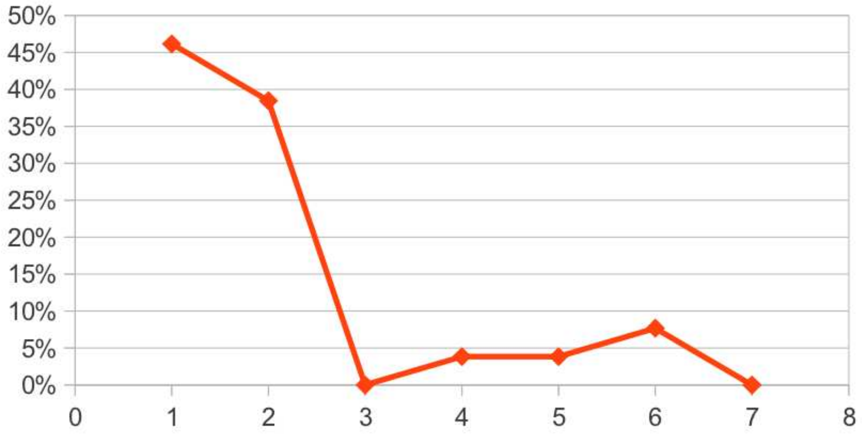
H2 is a relational database management system written in Java. It can be embedded in Java applications or run in the client-server mode, and it supports a subset of the SQL (Structured Query Language). It is possible to create both in-memory tables, as well as disk-based tables, so tables can be persistent or temporary.

This Benchmark executes a JDBC bench-like in-memory Benchmark, executing a number of transactions against a model of a banking application.

Shared Objects

Figure 3.12: Sampling Distribution of H2 Shared Objects

Owner Threads	Number of Nodes	%
1	12	46%
2	10	38%
3	0	0%
4	1	4%
5	1	4%
6	2	8%
7	0	0%
tot	26	



In figure 3.12 is shown the distribution of the sharing level. Given the structure of the Benchmark, which is a DBMS, we suppose that the presence of few object shared by almost all threads is due to connection objects, which share the connection to the DBMS with all the threads.

H2 consist of about 90% of the total of the trace, so an optimization of it's behavior is almost as optimizing the entire trace.

Hot Paths

In figure 3.11 we can see the *Hot Paths* graph, which is characterized by a strong presence of a sequence of two objects: *29532276* followed by *29623436*.

This sequence is probably due is due to the strong presence of synchronized methods that call other synchronized methods inside them, and it is thoroughly analyzed in section 3.6.

In particular in that section is proposed an algorithm which speculates on the presence of the chain in question, and optimizes a small part of the executed code.

If the speculation proves to be incorrect, the algorithm will make a switch to a much slower algorithm.

3.5.2 Luindex Benchmark

Description

The Luindex Benchmark uses Lucene to indexes a set of documents: the works of Shakespeare and the King James Bible.

Apache Lucene is a free/open source information retrieval software library, suitable for any application which requires full text indexing and searching capability.

Lucene has been widely recognized for its utility in the implementation of Internet search engines and local, single-site searching.

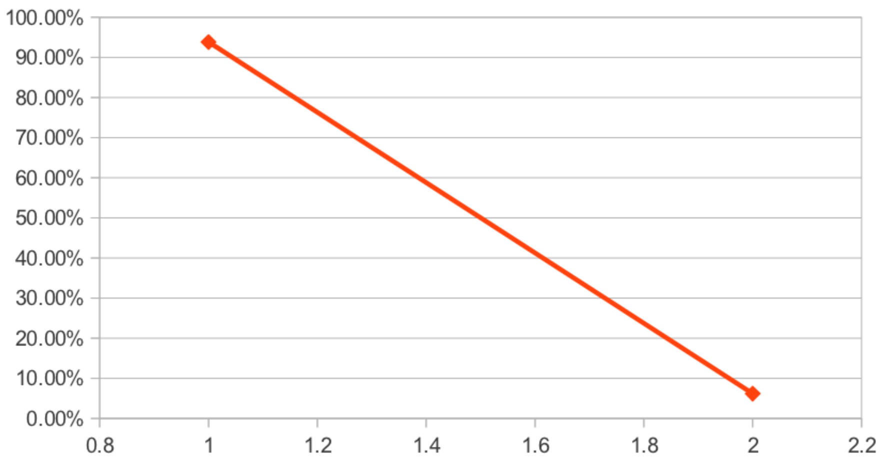
At the core of Lucene's logical architecture is the idea of a document containing fields of text. This flexibility allows Lucene's API to be independent of the file format, so Lucene can index text from PDFs, HTML, Microsoft Word, and OpenDocument documents, as well as many others.

Shared Objects

In figure 3.13 we can see that Luindex has only two principal threads running: this behavior is due to the main activities of this Benchmark, that is indexing a set of documents coming from two main sources.

Figure 3.13: Sampling Distribution of Luindex Shared Objects

Owner Threads	**Number of Nodes**	**%**
1	61	93.85%
2	4	6.15%
tot	**65**	



Only few objects are shared by all two threads: these objects are probabily connection and other objects used for execute queries and retrieve informations.

Luindex consist of about 1% of the total of the trace.

Hot Paths

In this Benchmark we can found a good presence of a Hot Paths, well-divided among themselves almost to follow the thread that owns them.

There is a little problem inside these chains: two of the most important objects are are listed in reverse order in the two paths.

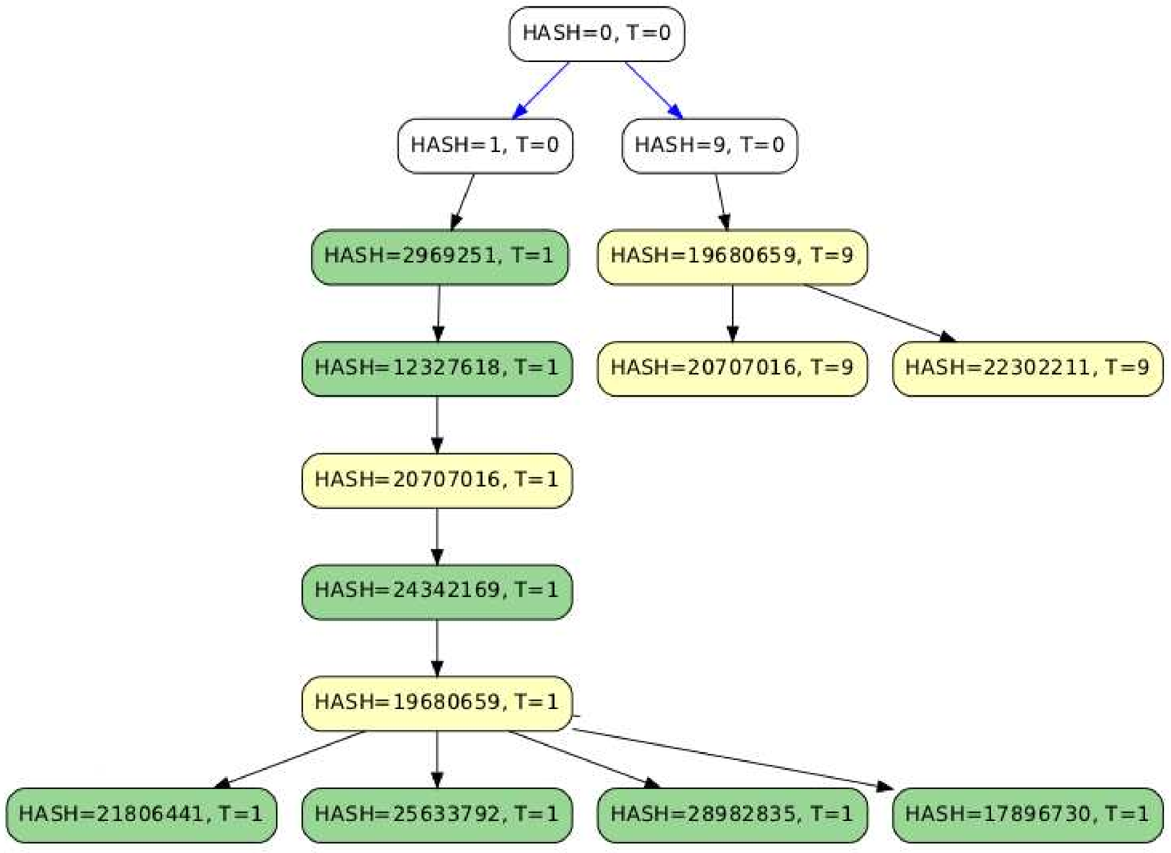
This feature prevents applying the optimized algorithm because it does not favor any speculation on that pair of objects.

3.5.3 Avrora Benchmark

Description

This Benchmark simulates a number of programs run on a grid of AVR microcontrollers.

Figure 3.14: Hot Path Tree of Luindex Benchmark

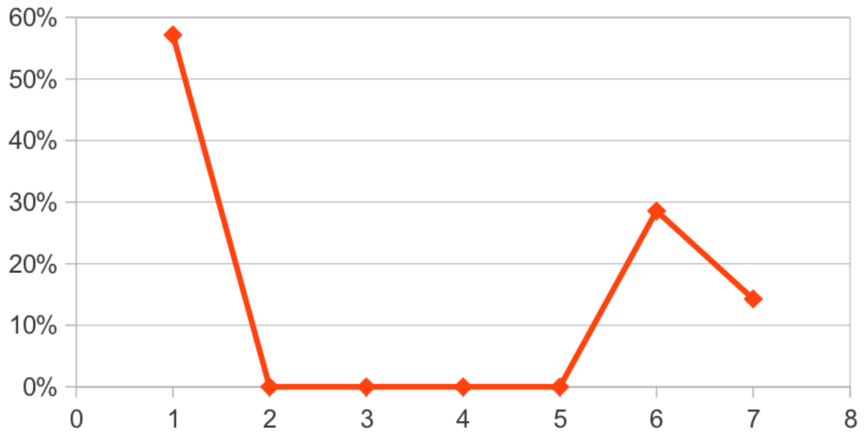


The AVR is a modified Harvard architecture 8-bit RISC single chip microcontroller which was developed by Atmel in 1996, and it was one of the first microcontroller families to use on-chip flash memory for program storage, as opposed to one-time programmable ROM, EPROM, or EEPROM used by other microcontrollers at the time.

Shared Objects

Figure 3.15: Sampling Distribution of Avrora Shared Objects

Owner Threads	Number of Nodes	%
1	4	57%
2	0	0%
3	0	0%
4	0	0%
5	0	0%
6	2	29%
7	1	14%
tot	**7**	



Avrora has a good part of locking-objects shared by all threads, and the remaining group of objects is shared by only one thread.

The fist ones are common objects for all threads, used to initialize the grid of AVR microprocessors or used in all the simulations, probabily to access particulary shared data between simulations.

The remaining set of objects, shared by a single thread, is very likely a set of objects used in individual simulations and accessed only by the thread that initiated the particular simulation.

Avrora consist of about 4% of the total of the trace.

Hot Paths

In the Max Path Tree of the Avrora Benchmark (fig. 3.16) we can see that there are no possible optimizations based on speculation of objects' chains, indeed in this tree there are no objects children of other objects, but all objects depend directly on the thread.

This does not mean that there are no locking chains of considerable depth, but among them there are no chains of such importance as to justify an optimization.

3.5.4 Lusearch Benchmark

Description

The Lusearch Benchmark uses Lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible.

Shared Objects

In figure 3.17 we can see that Lusearch has only few objects shared by all threads: these objects are probabily connection and other objects used for execute queries and retrieve informations.

An high number of objects are owned by a single thread: this is justified by the behavior of the Benchmark, which seeks a large number of keywords on a set of data.

Lusearch consist of about 5% of the total of the trace.

Hot Paths

As in Avrora, in the Lusearch Benchmark the Max Path Tree (fig. 3.18) is not useful if we want made some optimizations based on speculation of objects' chains, indeed in this tree there are no objects children of other objects, but all objects depend directly on the thread.

This does not mean that there are no locking chains of considerable depth, but among them there are no chains of such importance as to justify an optimization.

Figure 3.16: Portion of Max Path Tree of Avrora Benchmark

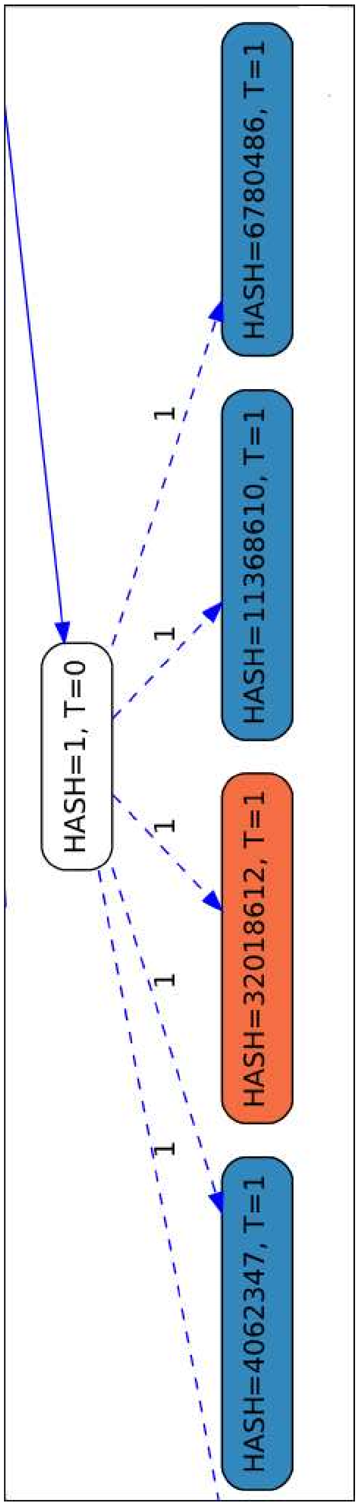


Figure 3.17: Sampling Distribution of Lusearch Shared Objects

Owner Threads	Number of Nodes	%
1	769	99.61%
2	0	0.00%
3	0	0.00%
4	2	0.26%
5	1	0.13%
tot	**772**	

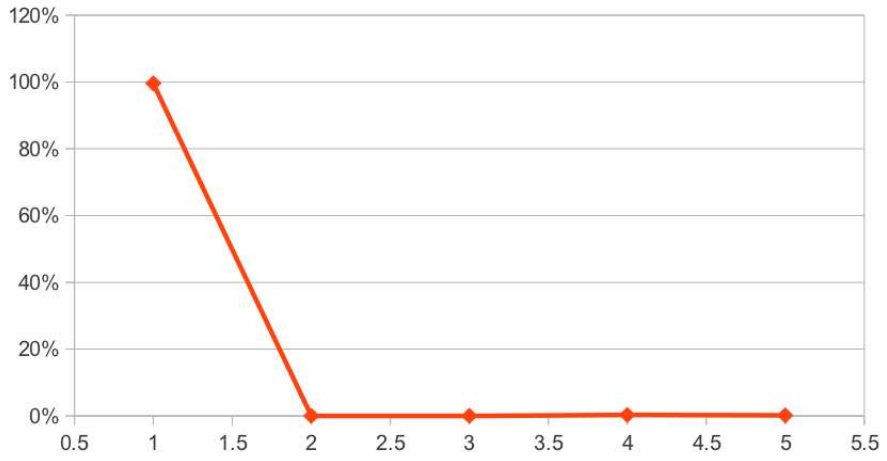
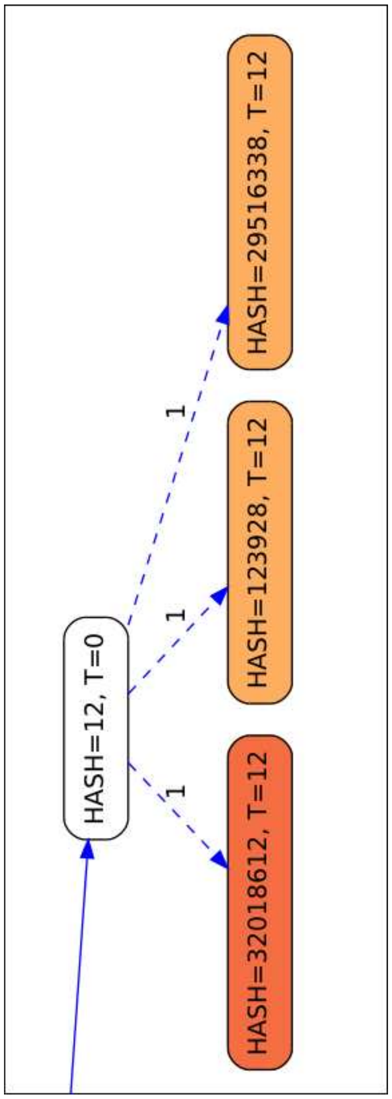


Figure 3.18: Max Path Tree of Lusearch Benchmark



This behavior is probably due to the fact that the Benchmark lacks synchronized methods that when executed, inside their code, they call methods of the same type.

3.6 Algorithm Proposal

In section 3.4.6 we calculated the *Hot Paths*, showed in figure 3.11 and we mentioned their importance to optimize the synchronization between threads.

In following sections we propose this algorithm, with its main characteristics and problems that can emerge, since this algorithm will exploit the Hot Paths and not always a sequence proves to be consistent.

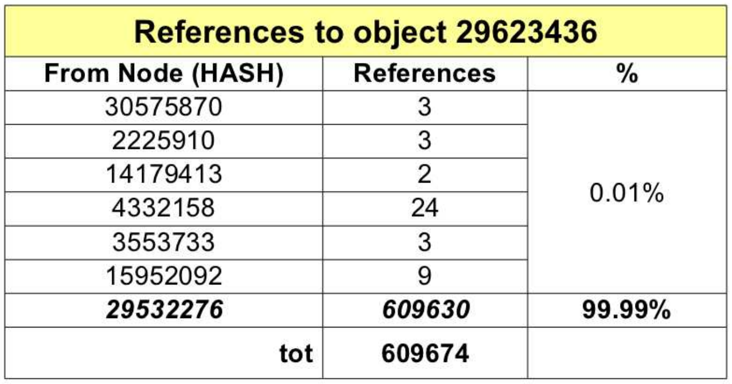
This algorithm tries to avoid synchronization instructions whenever the algorithm knows a priori a particularly sequence of nested locks: when a thread locks the first object, this algorithm give it the lock of the second object, without checking any locking table.

In this way it speculates on the structure of the chains and so it may avoid additional synchronization operations.

We will see that the exploit on this sequence affects a little portion of code, but in our analysis the speculation made on the sequence lead to an algorithm switch in very few cases.

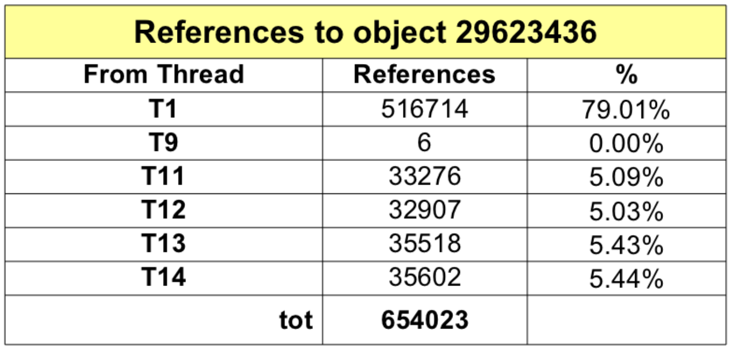
Every assumption is made considering only the H2 Benchmark, because it is the only one with desired behaviors and it consist of about 90% of the whole trace.

Figure 3.19: References to node 29623436 from other nodes



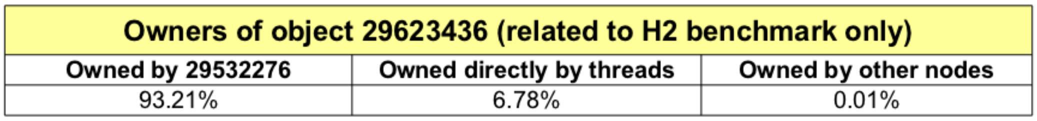
References to object 29623436		
From Node (HASH)	**References**	**%**
30575870	3	0.01%
2225910	3	
14179413	2	
4332158	24	
3553733	3	
15952092	9	
29532276	***609630***	**99.99%**
tot	**609674**	

Figure 3.20: References to node 29623436 from all threads



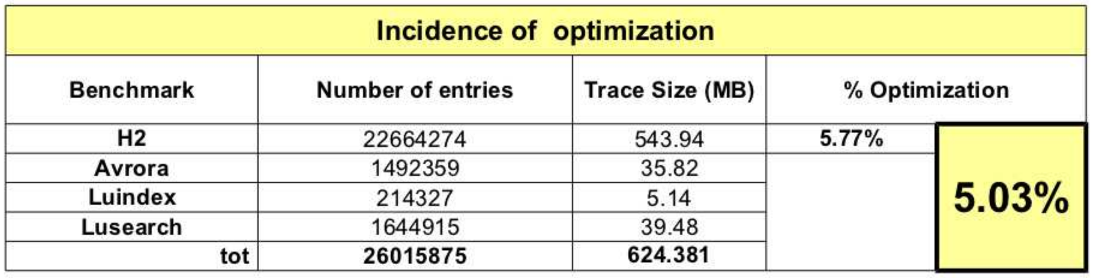
References to object 29623436		
From Thread	**References**	**%**
T1	516714	79.01%
T9	6	0.00%
T11	33276	5.09%
T12	32907	5.03%
T13	35518	5.43%
T14	35602	5.44%
tot	**654023**	

Figure 3.21: % of references to node 29623436 listed by type



Owners of object 29623436 (related to H2 benchmark only)		
Owned by 29532276	**Owned directly by threads**	**Owned by other nodes**
93.21%	6.78%	0.01%

Figure 3.22: % of optimization obtained by exploiting Hot Paths



Incidence of optimization				
Benchmark	**Number of entries**	**Trace Size (MB)**	**% Optimization**	
H2	22664274	543.94	**5.77%**	**5.03%**
Avrora	1492359	35.82		
Luindex	214327	5.14		
Lusearch	1644915	39.48		
tot	**26015875**	**624.381**		

3.6.1 Main Characteristics

The figure 3.11 shows the importance of the sequence *29532276 - 29623436*, which is repeated a lot of times and consist of two nodes with a very high level of sharing by threads.

The algorithm tries to avoid synchronization instructions whenever the algorithm finds the object with hash *29623436*, knowing a priori that the current thread already owns the object with hash *29532276*.

In this way it speculates on the structure of the chains and so it may avoid additional synchronization operations.

The speculation on this sequence affects approximately 5% of the entries, considering all the Benchmarks, and may lead to a gain in performance for that portion of code.

3.6.2 Issues

Unfortunately not everything is rosy, in fact, there are other sequences and it is possible that the object *29623436* is preceded by another object different than the one with hash *29532276* or that it is directly accessed by a thread.

In this case the algorithm must make a switch, leaving the control to lower level algorithm.

In figure 3.19 we can see that the % of other objects different than the one with hash *29532276* that own node *29623436* is very small, so the number of switch due to a wrong hypothesis of sequence is very small.

In the same way in the figure 3.21 is shown that the % of direct accesses to the object with hash *29623436* from the thread is slightly more than 6%.

In these cases the algorithm should not need a switch, but when an access to object *29623436* occurs, a set of bits may advise the algorithm if the threads accesses the object by itself.

4 Conclusions

The algorithm which do these optimizations, explained in section 3.6, acts in order to optimize the synchronization between threads in particular cases, that is when a known sequence of lock-unlock is executed.

In section 3.6 we made some hypothesis about a particular sequence composed by two *Hot* objects, one followed by another, and we suppose to create an algorithm that exploits this sequence.

This sequence affects approximately 5% of the total of the executed code: although there is a small portion of code that is optimized, it should be taken in account that the variants of the sequence, due to other objects that precede the second node, are present in very small percentage.

This ensures that the number of switches between optimized and not optimized algorithms, due to incorrect speculation on the presence of known sequence, are in a very small number, ensuring almost completely the execution of the optimized algorithm in the presence of the second object.

We can not estimate a priori how this optimization could speed up the entire run, because this has to be obtained by performing the analysis with the algorithm already running.

So the next step to do is to develop this algorithm, using the development environment made within this work, but there are some issues to take into account.

The first problem regards how to supply the algorithm a set of chains that can be optimized, since it is not possible performing a complete analysis on the code before executing it.

A possible solution consist of making the system able to self-learning which chains must be optimized, during the early stages of the execution.

Another problem is how to perform very fast switches between the optimized algorithm and slower ones, when the speculations reveals itself wrong.

Moreover, it is possible to compare the execution running time of the standard algorithm and of the optimized algorithm, obtaining a further estimate of the improvement obtained.

Bibliography

[1] Deadlock. `http://en.wikipedia.org/wiki/Deadlock`.

[2] DOT language. `http://en.wikipedia.org/wiki/DOT_language`.

[3] Java Wait() method. `http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/Object.html#wait(long)`.

[4] POSIX Threads. `http://en.wikipedia.org/wiki/POSIX_Threads`.

[5] Recursive Locks. `http://en.wikipedia.org/wiki/Reentrant_mutex`.

[6] Advanced Linux Programming. `http://www.advancedlinuxprogramming.com/`.

[7] E. Armstrong. Hotspot: A new breed of virtual machine. *www.javaworld.com*, 1998.

[8] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for java. *Proceedings of the ACM SIGPLAN 1998 conference on Programming Languages Design and Implementation*, 1998.

[9] Byte Code Engineering Library. `http://commons.apache.org/bcel/`.

[10] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, 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. pages 169–190, October 2006.

[11] T.H. Cormen. *Introduction to algorithms*. MIT electrical engineering and computer science series. MIT Press, 2001.

[12] DaCapo Benchmark Suite. `http://www.dacapobench.org`.

[13] Johnson Vlissides Gamma, Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.

[14] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. China Machine Press, 2008.

[15] Maurice Herlihy. Wait-free synchronization. *ACM*, 1991.

[16] Java Bytecode instruction listings. `http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings`.

[17] Java ClassLoader. `http://en.wikipedia.org/wiki/Java_Classloader`.

[18] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. *OOPSLA '02 , IBM Research, Tokyo Research Laboratory*, 2002.

[19] S.C. Reghizzi. *Formal Languages and Compilation*. Texts in computer science. Springer, 2009.

[20] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *SIGPLAN Not,*, 2006.

[21] A.S. Tanenbaum and A.S. Woodhull. *Operating systems: design and implementation*. Pearson Education International, 2009.