SAPIENZA
UNIVERSITÀ DI ROMA

A lock-free buddy system for scalable memory allocation

Facoltà di Ingegeneria dell'Informazione, Informatica e Statistica

Corso di Laurea Magistrale in Master of Science in Engineering in Computer Science

Candidate

Andrea Scarselli
ID number 1531508

Thesis Advisor

Prof. Bruno Ciciani

Co-Advisors

Prof. Francesco Quaglia
Dr. Mauro Ianni
Dr. Romolo Marotta

Academic Year 2016/2017

Thesis defended on 25 October 2017
in front of a Board of Examiners composed by:

Prof. Maurizio Lenzerini (chairman)
Prof. Roberto Beraldi
Prof. Vincenzo Bonifaci
Prof. Bruno Ciciani
Prof. Francesco Delli Priscoli
Prof. Giuseppe Oriolo
Prof. Silvio Salza

A lock-free buddy system for scalable memory allocation
Master thesis. Sapienza – University of Rome

© 2017 Andrea Scarselli. All rights reserved

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: October 23, 2017

Author's email: andreascarse@gmail.com

Dedicato alla mia famiglia,
che mi ha sempre sostenuto

Abstract

In the last years the time of easy performance gaining is ended due to a physical constraint called Power Wall. In this scenario, the large diffusion of shared-memory multi-core machines offers a new opportunities to face the increasing demand for improved performance. Anyhow, common lock-based synchronization techniques could be deleterious for performances and frustrate the presence of the increasing number of cores.

In order to optimize parallel execution, the non-blocking synchronization paradigm, based on the exploitation of Read-Modify-write (RMW) instructions, was born. This new technique requires a deep knowledge of the underlying hardware capabilities and on the assumptions that can be made on it.

On the other hand, the memory allocation problem is still relevant to support fast execution of both system and user applications.

An efficient allocator is required to avoid memory requests to become a bottleneck in high-performance scenarios characterized by large amount of processes. Moreover, requests could be very different depending on the target application; a good memory allocator has to works well in as many as possible scenarios.

In this work I present the design and the implementation of a lock-free buddy system memory allocator based on a binary tree structure. The proposed algorithm offers great performances in highly parallel machines and it is very memory efficient, producing low data overhead to work.

The results obtained in the experiments confirm the actual scalability of this proposal and the effectiveness of the lock-free synchronization.

Contents

1 Introduction **1**
1.1 The Synchronization problem . . . 4
1.1.1 Memory consistency models . . . 4
1.1.2 Atomic instructions . . . 7
1.1.3 Mutual exclusion . . . 9
1.1.4 Progress conditions . . . 11
1.2 Correctness conditions . . . 13
1.2.1 Serializability . . . 13
1.2.2 Linearizability . . . 14
1.3 Memory management . . . 15
1.3.1 Typical problems and challenges . . . 15
1.3.2 Presentation of some algorithms . . . 16
1.4 NUMA architecture . . . 22

2 Lock-free allocator: a brief introduction **27**
2.1 NUMA integration . . . 29

3 The algorithm and the implementation **31**
3.1 The idea . . . 31
3.1.1 Allocation . . . 34
3.1.2 Free . . . 37
3.1.3 Lock freedom proof . . . 40
3.2 Practical improvements . . . 42
3.3 Optimized algorithm . . . 45
3.3.1 Allocation . . . 48
3.3.2 Free . . . 51
3.4 NUMA implementation . . . 55

4 Experimental data **57**
4.1 Twenty-eight levels . . . 58

4.2 Twenty levels 59
4.3 Sixteen levels 61
4.4 Time sharing 62

5 Conclusions **65**

List of Algorithms

1 CAS 7
2 Classical buddy system allocation 28
3 Classical buddy system free 28
4 Allocation simplified 35
5 Check parent simplified 36
6 Free simplified 38
7 Unmark simplified 39
8 Allocation 49
9 Occupy descendants 50
10 Check parent 51
11 Free 52
12 Free descendants 52
13 Mark 52
14 Unmark 54

List of figures

1.1 Progress taxonomy 13
1.2 Original schedule 14
1.3 Equivalent serial schedule 15
1.4 Example of a NUMA machine with a meshed topology 25

2.1 Basic representation of the data structure 29

3.1 Node's possible state 33
3.2 Free of the node 4, occupied and coalescing bits remains marked in the node number 1 38
3.3 Three levels organization 44
3.4 Four level organization 47
3.5 Allocation of node which is not a leaf (the red one) in the compact-level algorithm 48
3.6 NUMA implementation 55

4.1 Execution time with 28 levels. 60
4.2 Execution time with 20 levels. 60
4.3 Execution time with 16 levels. 61
4.4 Time sharing results. Twenty-eight levels. 63
4.5 Time sharing results. Twenty levels. 63
4.6 Time sharing results. Sixteen levels. 63

Chapter 1

Introduction

The increase of microprocessor performance, in the last years, has been less important respect to the past due to a law called "the power wall". It happens that the CPU's clock frequency is proportional to the power consumed; the problem is that we can not exceed a certain power threshold both for consumption reason and for physical reason.

However, respecting Moore's law, number of transistors doubles every two years. With these transistors, it is possible to create auxiliary circuits to improve overall performance; examples of this kind are those used to achieve Instruction Level Parallelism (ILP) with deeper pipelines and replicated components; and branch predictors for speculative processing.

Deeper pipelines imply to have multiple instructions running simultaneously in the CPU: with pipelining, instructions are divided in stages (in a basic pipeline we have at least the stages: Fetch, Decode, Execution, Memory Access and Write Back) and each of this stage will be, at a certain time, processing an instruction; the more deep is the pipeline, the more instructions will make progress in a single clock cycle.

With replicated components it is possible to have more than one instruction in each pipeline stage; this improvement is possible by replicating entirely the pipeline (superscalar architectures) or by replicating only certain components.

With ILP we can have the index IPC (Instruction Per Clock cycle) greater than one: this tells us that we can commit more than an instruction for each clock cycle.

There are basically two manners to implement ILP in presence of replicated components: **static parallelization of the execution**, in which decisions of how to implement it are taken at compile time and **dynamic parallelization of the execution**, in which those decisions are taken at runtime. The major problem of this kind of approach is the *ILP wall*, which represents the difficulties to find a greater number of operations that could be run in parallel, i.e. that are independent from previous operations.

With branch prediction it is possible, in a pipelined architecture and in presence of conditional jumps, to try to guess which will be the next instruction to execute, in order to fetch it without waiting to know which is the next operation to do (this would mean to wait the previous instruction to terminate at least the execution stage).

This kind of sophisticated technique actually was not sufficient to satisfy the increasing need of performances. During the last decades, the higher number of transistors has been used to develop multi-core CPUs. This kind of architecture allows a big performance improvement, maintaining almost equal the clock frequencies. In fact, often a big problem can be faced decomposing it into multiple smaller and independent sub-problems that could be solved in parallel.

In this kind of architecture new problems, that arises because of some resources are shared among multiple cores, are to be considered. For example, due to the fact that the memory is shared, it could happen that a core finds it busy when it need it. If the core wants to write a result to the memory, to avoid the pipeline stall waiting the memory availability, a common hardware optimization was introduced: a store buffer in each CPU core. With the store buffer, when a core has to write to the memory, if it is not available, it stores the value in the local buffer in order to flush all the pending write when the memory become available. This leads to a good performance enhancing due to the less stall of the pipeline but it will also arise new problems that will be discussed in this work.

When we talk about parallel programming, we usually talk about an architecture known in literature as **M**ultiple **I**nstruction **M**ultiple **D**ata [Flynn, 1966]. In this kind of architecture, a set of instructions are executed simultaneously on a set of data; this is possible thanks to the utilization of multi-processors/multi-cores machine.

Flynn classified two more architectures for parallel programming:

- **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD): in this kind of architecture the same instruction is simultaneously executed on a set of data; this is possible in all the modern CPU thanks to "vectorial operations" and, moreover, it is largely adopted with the Graphic Processing Unit;

- **M**ultiple **I**nstruction **S**ingle **D**ata (MISD): in this kind of architecture multiple instructions are executed simultaneously on the same data; this type of architecture is actually a theoretical one; it has never become a real commercial product.

Moreover, he classified the standard, non-parallel architecture as **S**ingle **I**nstruction **S**ingle **D**ata (SISD) in which a single operation at a time is executed on a single data.

Parallel programming requires knowledge of what are the peculiarities of the underlying architecture. A program that runs well in a certain machine, could be wrong on another one; in fact, correctness of programs depends on what memory consistency model is implemented by the architecture.

Even if, as aforementioned, we aim to decompose the problem into many independent parts, at some point a synchronization between threads will be necessary.

Synchronization between parallel processes/threads is one of the bigger challenge of computer programming; historically it was done by exploiting the concept of lock: the computation halts waiting the synchronization between threads. Synchronization usually denies the possibility to see linear speedup augmenting the core number. To try to achieve a good performance gaining by augmenting the number of cores, in our days, as we will see, more sophisticated synchronization primitives are available.

The speedup is defined as the performance improvements achieved augmenting core number. We have two laws that bounds the theoretical speedup (it is important to notice that these laws give us only an upper bound since they do not consider any synchronization cost):

1. the Amdahl law, which is the most famous one, tells us how the execution time varies augmenting the number of cores and maintaining fixed the workload. The law says that:

$$Speedup = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$

   where p represents the number of processors and α is the serial part of the program (i.e. the part that can not be parallelized). It is important to notice that, using an infinite number of processors we have that the speedup is bounded to:

$$Speedup = \lim_{p \to \infty} \frac{1}{\alpha + \frac{(1-\alpha)}{p}} = \frac{1}{\alpha};$$

2. Gustafson's law tells us the speedup that we have augmenting the number of cores and the workload, maintaining fixed the execution time. Given the initial workload *W*, we can obtain the scaled workload *W'*, obtained augmenting the number of cores:

$$W' = \alpha W + (1-\alpha)pW$$

   The law tells us that the speedup is equal to:

$$Speedup = \frac{W'}{W} = \alpha + (1-\alpha)p.$$

Other important parameters are the efficiency E and the scalability. The first

one is simply defined as:

$$E = \frac{speedup}{\text{number of processors}}$$

Scalability tells us how the program reacts when the number of processes or the problem size increases. We have:

1. *strong scalability* if the efficiency remains constant increasing the number of cores and maintaining fixed the problem size (i.e. the speedup is linear in the number of cores);

2. *weak scalability* if the efficiency remains constant increasing in the same manner number of cores and the problem size.

In this work, parallel programming will be used to develop a memory allocator based on the buddy system algorithm.

1.1 The Synchronization problem

As said, developing parallel programs require to take in mind how to synchronize multiple threads.

In the first subsection, I will introduce memory consistency models and the atomic instructions, which are often used for synchronization purposes in architecture which does not guarantee instruction ordering; in the second subsection, I will introduce the mutual exclusion, which is a classical technique used to achieve synchronization and finally, in the last subsection, I will talk about progress conditions.

1.1.1 Memory consistency models

Modern processors can execute operations in an order that could be different from the order in which instructions are written. This kind of behavior is totally uncontrollable by the programmer and, moreover, can vary from an execution to another. This kind of situation is born for performance reason, in fact, it can happen that the actual instruction refers to a data that is not in the cache; the processor, instead to stall, waiting the memory to fill the cache, executes the first operation which has all the required data into the cache.

This kind of behavior, although not intuitive, actually does not affect single threaded programs; however, it can cause synchronization problems in a multi-threaded environment.

With this architecture, it is important to formally define how operations can be inserted in the global memory, which is shared among the multiple core/processors.

Memory consistency models are used to define what the programmer have to expect from the underlaying architecture about operations ordering.

While programming a multi-thread program it is important to take in mind which is the memory consistency model implemented by the specific architecture, in order to avoid obtaining unexpected (often wrong) executions. In particular, we can have four different types of reordering:

- store-store reordering: it happens if a store S1, which has to be done before S2, is committed by the write buffer after S2. This could happen, for example, in case if S1 gets a cache miss: in this scenario, to avoid pipeline stall, CPU executes S2 (if it gets a cache hit). This kind of reordering could be avoided by using a FIFO write buffer that guarantees that stores are flushed in the same order as they entered the CPU;

- load-load reordering: it is similar to the store-store reordering and it also happens due to cache miss;

- load/store and store/load reordering: it happens if two subsequent operations of load and store are executed in an inverse order. For correctness conditions, obviously it can be done only if the store and load are referred to two different addresses.

Respecting the constraint that load/store and store/load reordering are admitted only in the case in which they refer to two different addresses it is sufficient to guarantee correctness of single-thread programs.

To avoid reordering problems it happens that in some architectures operations are reordered but they are committed (retired) in the original order: it happens that the actual order in which the operations are executed is totally invisible and not significant.

To formally define memory consistency models, we need the concepts of:

1. program order $<_p$ that is a per-core total order. If a core has to do two memory operations, say O_1 and O_2: if $O_1 <_p O_2$ it means that the CPU logically executes (commits) O_1 before O_2;

2. global memory order $<_m$ that represents the order in which the operations performed by all the cores appears in memory.

Sequential Consistency

In [5] Lamport defines the concept of Sequential Consistency (SC). An execution satisfies sequential consistency if:

- all cores insert their loads and stores into the global memory respecting their program order, formally, referring to L(a) as a load on a memory address a and S(b) a store on the memory address b (which can be different or equal to a), we have that:

  1. $L(a) <_p L(b) \implies L(a) <_m L(b)$
  2. $L(a) <_p S(b) \implies L(a) <_m S(b)$
  3. $S(a) <_p S(b) \implies S(a) <_m S(b)$
  4. $S(a) <_p L(b) \implies S(a) <_m L(b)$

- it resembles that threads executes memory operations in an interleaved fashion and the interleaving of thread's operation seems arbitrated by some switch;

- every load gets its value from the last store happened before it (in global memory order) in the same address.

It is important to observe that, due to the store buffer, a great number of modern architecture does not support SC, they only guarantee a relaxed memory consistency model called Total Store Order (TSO).

Total Store Order

Total Store Order (TSO) is the actual memory consistency model guaranteed by well-known and widely used architectures like x86, x86-64 and some SPARC implementations.

This kind of architecture violates sequential consistency due to the store buffer implementation. In fact, delaying writing of the results in memory, can cause a store/load reordering that breaks the following SC conditions:

$$S(a) <_p L(b) \implies S(a) <_m L(b)$$

It can happen that

$$S(a) <_p L(b) \text{ AND } L(b) <_m S(a).$$

Although this kind of reordering does not lead to any problem in most of the applications (due to the fact that it is always guaranteed that a store/load reorder is only permitted if $a \neq b$), there are specific situation in which this reordering can not be admitted: what happens if a value, needed for synchronization purposes is not immediately seen by other threads? Synchronization becomes impossible.

To solve this problem, specific operations, called *FENCEs* are implemented in TSO architectures; executing this kind of operations, the store buffer gets flushed,

respecting the FIFO ordering of the pending writes; however, this lead to a significant performance worsening because the store buffer implementation become useless.

Weaker memory consistency

Some important architectures like IA64, ARMv7, POWER only guarantee the impossibility to have store/load and load/store reordering when they reference the same address. This kind of architectures arbitrarily reorder memory operations and it is possible only to control them by using FENCEs. However, maybe surprisingly, this kind of behavior, without using FENCEs does not affect at all correctness of the most applications.

1.1.2 Atomic instructions

Synchronization primitives usually exploit built-in atomic functions called Read Modify Write (RMW) operations. Such functions are architectural implemented and they permit a thread to atomically read, modify or act on the retrieved value and then write it back. If a thread executes an atomic function it has the guarantee that no other thread could access (neither in read mode, nor in write mode) the involved variable. This means that the access to a memory location is serialized.

An example of atomic instruction is the Compare And Swap (CAS); this operation needs three parameters (memory location, the old value and a new one) and it has the following semantic: if the memory location has the value "old value", the operation atomically substitutes the value with "new value" and it returns true; otherwise the memory location is not modified and the operation returns false. This atomic function is implemented in almost all the modern architecture like x86[1], SPARC, ARM etc. Pseudo-code can be find in algorithm 1.

Algorithm 1 CAS

```
1: procedure CAS(void* p, long old_val, long new_val) return bool
2:     atomically do:
3:     if *p ≠ old_val then return false
4:     *p = new_val return true
```

Other examples of widely used atomic instructions are:

- Test And Set: it atomically sets a variable to the value 1 and returns the old value of the variable;

[1] In x86 CAS is implemented with the instruction CMPXCHG. Writing high level code it is possible to use it exploiting built-in gcc compiler function ___sync_bool_compare_and_swap. There is a built-in compiler function for each atomic instruction.

- Fetch And ADD, Fetch And OR, Fetch And AND etc. : there is an atomic operation basically for all the arithmetic operations. They atomically take the old value of the variable and they perform on it the opportune operation, storing the result in the original memory location;

- Load-Link/Store-Conditional (LL/SC): they are a pair of atomic instructions (not implemented in x86) which have the following semantic: the load-link operation returns the value of a memory location; then the store conditional stores a new value on it if no one has modified the variable since the load-link. This is actually different from the CAS where it is only important that the actual value of the variable is equal to the one written in the parameter "old_val". This means that, if a process P reads a value x from a variable at a time *t*; a process P' writes the same value x on the variable at time *t'*; if P performs the compare and swap (CAS) at time *t"* with the old value equal to x it will be successful, instead, if P, at time *t*, read the variable with a load link (LL) and he tries to write the variable at time *t"* with a store conditional (SC), it will fail.

LL/SC was born to solve a well-known problem that arises using atomic instructions. This problem is known in literature as ABA problem; basically, we have it when it happens that:

1. a process P reads the value x in a variable "`var`";

2. a process P' changes the value of var from x to y;

3. any process different from P changes again the value of var to x;

4. P tries a CAS on `var`, passing as old value parameter the value x. The CAS is successful and returns true. However, the fact that the value of `var` has been changed could be meaningful in some applications and so it could happen that the successful CAS was wrong in respect to the logic of the application.

It can happen that this kind of behavior, in some applications, can lead to wrong states. Due to the fact that LL/SC operation (which definitely solve the ABA problem) is not implemented in all the architectures, the programmer, which is implementing an application that is sensible to the ABA problem and is forced to use CAS, has to implement some method to recognize updates in the memory location. The simplest way to reach this goal is to steal some bits to the variable and use them as a counter storing the epoch of the corresponding write operation; obviously this solution complicates the program and problems like the overflow of the counter must be kept in mind.

This kind of operations are used to implement all the efficient synchronization primitives and so they are actually very useful.

Read Modify Write Operations, actually call a FENCE after their execution to make the result visible in the global memory. As aforementioned, this have a negative performance impact, and, for this reason they have to be used with care, limiting their execution as much as possible.

1.1.3 Mutual exclusion

As mentioned before, to make correct parallel programs it is important that a process synchronize itself, i.e., it is necessary that each process see the same memory snapshot and it is mandatory that, if a process P is working on a shared variable, no other process P' modifies that variable, or, at least, if it happens, P has to be noticed of that immediately. Historically, the first solution to this problem was the mutual exclusion.

Mutual exclusion is coupled with the concept of *critical section*. A critical section (in the field of concurrent systems) is a piece of code in which are accessed portions of memory that are shared among some processes (and at least a process can write this shared memory). To guarantee consistency and correctness respect to mutual exclusion it is required a mechanism such that only a process at a time can execute the critical section. A software solution (which does not need any hardware support) used to implement mutual exclusion is the Baker's algorithm, which resembles the procedure done in a common Baker. However, this algorithm (like all the pure software solutions) is based on busy waiting. Since this approach fully loads the CPU it is unacceptable for performance and power consumption. For this reason, if we are in an environment where atomic instructions are supported, we can implement mutual exclusion by using:

1. lock: it is a shared, variable which can only assume two different values, e.g. 1, meaning that there is the possibility to execute the critical section and another value, e.g. 0, which says that there is not the possibility to execute that code. The atomic operation Test And Set is exploited: the process that is able to change the value of the variable from 1 to 0 obtains the lock and it is admitted to execute its critical section; other processes will try (for an undefined time, doing busy waiting) to obtain the lock. To ensure progress, the process which execute the critical section has to restore the variable's value from 0 to 1 when it finishes its critical work. This mechanism suffers of *starvation*: a process (or a set of them), for some reason, could be always locked trying to obtain the lock, while the other processes make progress;

2. semaphores: it is a shared, atomic structure in which the accesses are regulated by a FIFO queue. This structure admits two atomic operation which are: *wait()* and *signal()*. With the first operation, a process signals the willingness to enter the critical section. If someone else is already executing it, the process goes in the queue and it waits its turn. When this moment come, the structure wakes up the process. At the end of the execution of the critical section, the process has to do a *signal()* operation to permit the next process to enter the critical section.

All the modern operating systems supports locks and semaphores and relative APIes are available to exploit them.

Semaphores could be a good choice to implement mutual exclusion since, by exploiting the FIFO queue, solves the starvation problem. Moreover, with this approach a process which has to wait to enter a critical section, does not do busy waiting and this allows both power and CPU savings. However, semaphores could offer poor performances in a high-parallel environment due to the need to the operating system to sleep and wake up process, using signals, which are highly expensive operations.

In all the cases, writing algorithms based on mutual exclusion can lead to some important troubles:

1. it is a pessimistic approach: if a process P_1 obtains a lock, other processes, to take this lock have to wait that P_1 releases it. However, in practice, it could happen that if other processes had executed the critical section, there would be no conflicts. E.g: two processes that want just to read a shared variable; two processes that works on two different variables etc. To solve this problem sophisticated method were implemented: 2PL, fine grain locking and so on... however, these solutions complicate the code and can lead to write a lot of programming errors;

2. deadlock possibility: we have deadlock if a process, P_1, acquires a lock L_1 and another process, P_2, acquires a lock L_2. Later, P_2 requires the lock L_1 which is still busy with P_1 and P_1 requires L_2 which is still busy with P_2. In this case, we are in a condition where the two processes are waiting their self and we will not have progress. Some techniques have been developed to solve this problem like deadlock avoidance and deadlock detection and recovery;

3. if a process obtains a lock and it crashes before to release it; no more processes will be able to enter the critical section;

4. lock does not compose: if we have two correct pieces of code that use lock;

this does not imply that, if we merge the two pieces of code, the union will still be correct;

5. debugging could be very difficult because it could happen that a bug will appear only in some particular schedule, difficult to locate and reproduce;

6. the highly unpredictability of the runs, and the possibility of a great starvation, that we have by exploiting locks could be problematic in a lot of environment.

1.1.4 Progress conditions

Progress conditions tell us in which condition and with which guarantees a function (or method) is able to do its work, i.e. it does not halt. Progress conditions can be grouped with three different properties:

- Minimum or maximum progress: we say that we have minimum progress if, in a program, some method completes eventually; we have maximum progress if all the methods complete eventually;

- Dependent or Independent: a progress condition is dependent if it has to make some assumptions on the scheduler; independent if it does not require any guarantees from the scheduler;

- Blocking or non-blocking: a progress condition is blocking if it requires mutual exclusion; non-blocking otherwise.

In a multi-threaded environment progress is just a liveness condition: it tells us that something eventually will happen but it does not say anything about its correctness. Using both progress conditions and synchronization it is possible to have programs that runs (liveness) correctly (safety).

In the rest of this section are presented the six progress conditions.

Blocking conditions

Blocking conditions relies to the concept of mutual exclusion presented before.

Deadlock Free and Starvation Free The two blocking progress guarantees are *deadlock free* and *starvation free*. They are both dependent conditions because they require to the scheduler that, if a process P gets a lock, it will not be in a sleeping state forever, causing the inability to enter the critical section to the other processes.

Deadlock free is a minimum progress condition: it requires that some method acquires a lock eventually;

Starvation free is a maximum progress condition: it requires that every method eventually acquires the lock.

Non-blocking conditions

Obstruction free Obstruction free (like lock free and wait free) algorithms does not use lock to make the synchronization possible. An algorithm is said to be obstruction free if it has the following properties:

1. there is no lock (neither implicit, nor explicit);
2. there is maximal progress;
3. it is dependent from the scheduler and, in particular, it requires that threads run in isolation for a time long enough to complete the critical operations.

Practically this condition says that each function completes if it is executed in isolation.

Actually, there is the minimal progress counterpart, which is called **Clash free**. It has been proved that this progress conditions is weaker than obstruction free, however clash free algorithms have never been developed.

Lock Free An algorithm is lock-free if the following are satisfied:

1. there is no lock (neither implicit, nor explicit);
2. there is minimal progress;
3. independence from scheduling.

We can say that an algorithm is lock-free if there is always the condition where at least a thread makes progress; if a method does not make progress, it means that another process is surely making progress.

Wait Free An algorithm is said to be wait-free if we have:

1. algorithm without lock (neither implicit, nor explicit);
2. maximal progress;
3. independence from scheduling.

Wait-freedom implies lock-freedom. Wait-free algorithms are very hard to be implemented and they risk to suffer of poor performance compared to lock free and blocking counter parts. However, the recent publication of [33] and [32] allowed the creation of efficient wait-free data structures and these kinds of algorithms are used today.

A summary of progress conditions is reported in figure 1.1. In the left column, we find non-blocking progress conditions, on the right instead, we have the blocking; in the first row, we found conditions that gives us maximal progress warranty, minimal progress in the second. Moreover, blue cells highlight independent conditions; in the yellows cells, there are the dependent.

	Non blocking		Blocking
Maximal progress	Wait free	Obstruction free	Starvation free
Minimal progress	Lock free	Clash free	Deadlock free

Figure 1.1. Progress taxonomy

1.2 Correctness conditions

In a concurrent environment processes use and exploit shared object. These objects could be accessed concurrently and we need mechanisms to preserve integrity of the operations (also called transactions) that manipulate them. We can have different conditions that specify what is correct and what is not. When we talk about correctness conditions it is necessary to define the concept of history (also called schedule in literature).

We define the history H as a set of invocations and responses relative to a certain shared object; an history H is equivalent to an history H' if they have the same set of operations. I will briefly present two important correctness conditions that are serializability and linearizability.

1.2.1 Serializability

A notion of serializability is introduced in [44]. First, we have to define a *serial schedule* S as an history where, all the operations of a single transaction T, are executed before of the first instruction of another transaction T' in S. This happens if, for example, transactions are not subject to interleaving. A schedule S is serializable if, for each possible initial state of the database, results of S are equal to at least another equivalent schedule S', which is serial. Observing serializability definition we can notice that it is intrinsically a blocking property [26]. Serializability is commonly

used in Database Management Systems (DBMS).

1.2.2 Linearizability

A notion of Linearizability is introduced in [26]. This condition is actually used to demonstrate correctness of concurrent objects.

An history is sequential if each invocation immediately receives a matching response and after that, instantly we have a matching invocation. A non-sequential history is called concurrent.

An history H is said to be linearizable if:

1. invocations and responses could be reordered in order to have an equivalent history H' which is sequential;

2. the obtained history H' is correct respect to the sequential definition of the object on which the operations are done;

3. if a response precedes an invocation in H then the same order is maintained in H'.

Objects satisfying this correctness condition give illusion that the operations are done instantaneously, in a certain point between the invocation and the response.

The actual difference from serializability is that, given a specific schedule S, in order to respect serializability it is sufficient to find a serial, equivalent schedule S'. To satisfy linearizability we have to find a schedule S" which is equivalent, serial and also respects the ordering condition.

This example shows a schedule, referred to a FIFO queue, which is serializable but it is not linearizable:

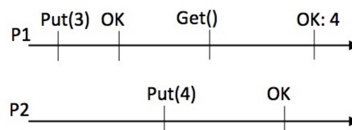

Figure 1.2. Original schedule

In this example:

1. *put(x)* represents an insertion of an element of value x in the queue;

2. *get()* represents the operation of the extraction of the oldest value (respecting the semantic of a FIFO queue);

3. "OK" represents a response which does not contain payload;

4. "OK : x" represents a response with payload x.

One possible equivalent, sequential schedule S' is the one in figure 1.3. With S' we see that the schedule in the example satisfies serializability but it does not satisfy linearizability. It happens that, due to the original ordering, operation "put(3)" must be the first: this make impossible to find a schedule that respect the definition of a FIFO queue in which the get returns 4.

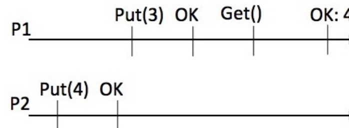

Figure 1.3. Equivalent serial schedule

1.3 Memory management

Memory management is one of the most important problem to deal with when it is needed to ensure high performances. Due to the fact that application level code, during its execution, often allocates and deallocates memory, its execution will never be fast enough if the memory allocator represents a bottleneck.

In our days memory allocation is even more important, in fact, with the advent of high parallel machines it is mandatory to meet memory requirements in a time that is as fast as possible. This happens because memory access is a serialization point and, if one process takes a lot of time to do a memory request, it will slow down all the other threads that are attempting to access the memory too.

A first solution to face this problem is offered by this problem **N**on **U**niform **M**emory **A**ccess (NUMA) machines. With this kind of machines a set of cores can access a certain portion of physical memory with lower latency; the result is that the contention is reduced because processes are smeared in multiple serialization point.

In the first section are presented typical problems and challenges that there are thinking about developing a memory allocator; subsequently are presented some algorithm developed during the years, highlighting their concurrency conditions and finally an overview on the NUMA architecture is provided.

1.3.1 Typical problems and challenges

A good general-purpose memory allocator has to satisfy at least the following requirements:

- reduce internal and external fragmentation;

- require little overhead for meta-informations needed to remember necessary informations like, for example, which blocks are free and which are not, the dimension of the blocks, etc;
- allocate and deallocate memory as fast as possible;
- gives memory to the calling process if and only if the memory is actually available, avoiding wrong situations in which a chunk of memory is given to more than one process;
- the allocator must be scalable in order to efficiently support highly parallel machines.

Internal fragmentation

In the context of memory allocation, we have internal fragmentation when a request of memory of x bytes is satisfied allocating a memory chunk of size $y > x$. This happens when the memory is pre-partitioned in blocks and an allocation has to allocate an entire block. This problem could be mitigated using blocks of variable length and satisfying a request with the smaller available block.

It could be solved by allocating blocks of memory of the exact dimension required by the allocation; however, this kind of algorithms could offer poor performance and, moreover, after the execution of a certain number of alloc/free they could have a big problem with external fragmentation.

External fragmentation

We have external fragmentation when a request of x bytes could not be satisfied although there are x (or more then x) available bytes. This happens if the free memory is partitioned in a manner such that the x free bytes are not contiguous. This problem could be solved by, periodically, compacting the free blocks: this means to move all the occupied block to the initial part of the memory; however, this operation is very costly (it has cost O(n) where n is the size of memory) and it is actually difficult to be implemented, e.g. physical address of the variable need to be changed. To mitigate this problem is sufficient to coalesce (merge) two contiguous free block in order to have the bigger free blocks possible.

1.3.2 Presentation of some algorithms

Memory allocation problem was born with the advent of multi-programmed systems. In fact, with this organization, a single instance of physical memory has to be divided for an arbitrary number of processes, in order to store the processes workspace. The

first memory management algorithms used a mechanism called "static partitioning, with multiple partitions": this algorithm a-priori divides the memory in a certain number of partitions (of fixed or variable size) and, it dedicates a single partition for each process; the process will be fully loaded onto the memory when it is launched. This mechanism gives the possibility to have multi-programmed systems but raises a certain number of problems:

- the number of processes that the system could manage is fixed and it is equal to the number of memory partitions;
- each process will fully occupy the partition, also in the case where required space is smaller. This raises the already discussed internal fragmentation problem;
- using *variable size partitions*, deciding to use a queue of process for each block size, it is possible also to incur in the external fragmentation problem, in particular, it could happen that a process requires a slot of memory of x Byte, which is not free; the process will hang also in the case where a block of size $y > x$ is free; however, this mechanism actually mitigate the internal fragmentation problem.

Second family of memory management algorithms used a mechanism known as "dynamic partitions". With this mechanism, when a process gets launched, it is fully loaded onto the memory in a new partition, expressed created for it, with the same dimension of the loaded process. In this manner the internal fragmentation problem is definitely solved; however, the fact that when the process terminates, it releases its memory where it is, poses a big problem of external fragmentation: in fact, after a great number of allocation/deallocation the memory will be more and more fragmented. To solve this, this family of allocators usually uses a periodic compaction strategy.

Third family of allocators are those based on "pagination". With this kind of algorithms, a process that needs to be launched, is divided in a certain number of fixed size "pages". In this manner, the process could be loaded in not contiguous portions of memory. This approach definitely solved the external fragmentation problem. However, this solution suffer of internal fragmentation, in particular, the maximum internal fragmentation that we could have is equal to $PAGESIZE - 1B$ obtained when a process need only a byte from a new page; however, typical pagination algorithms uses pages of 4KB and so the wasted memory with internal fragmentation can be considered negligible. This kind of algorithms permitted the born of the "virtual memory" concept. With this methodology, the process will not be fully loaded onto the memory when it is launched, instead, a page of the process,

will be loaded in when it is necessary. Moreover, if the system needs memory, it can move back on the disk a page of a process; it will reload it when it will be necessary again.

With this kind of technology, memory allocation seems to have reached a good level of performance and flexibility but, this kind of memory allocators, are not efficient if we need, for performances reasons, contiguous physical memory like the case of an operating system kernel.

For this, particular, kind of allocation we use a methodology called Buddy System. In a buddy system, memory allocations are possible in block of size $2^i * PAGESIZE$; the algorithm basically works in the following manner:

- the larger block has a dimension equal to the maximum allocable size;
- when a process requires x bytes of memory, x is rounded to y, the nearest power of two;
- free lists are used to manage free blocks. There is a free list for each possible allocable size;
- if there is a block of size y, the block is delivered;
- if not, a bigger block is divided by two, until it reaches the size y;
- when the allocator sees that there are two adjacent blocks of size 2^i, they are merged in order to form a bigger block of size 2^{i+1}. This is done in order to reduce external fragmentation.

Linux buddy system [15] utilizes 11 free lists, in this manner it is possible to allocate from 1 page to a maximum of 10 pages, i.e., the linux buddy system can serve at most allocation of 4MB.

Modern memory allocator with lock

In this section are presented three famous memory allocators that use locks in order to guarantee correctness in concurrent environment. [**?**], the standard glibc allocator [1] and tcmalloc [3].

Hoard has been considered the best memory allocator for a long time. In Hoard, there is a global heap and 2P local heaps, where P is the number of processors, in order to better support parallel accesses. A thread, which is executing Hoard, and that wants to allocate new memory, will access the local heap H, where H is obtained giving the process ID (notice that the function uses the process ID, not the processor ID) to a hashing function. This hashing function is called each time to support processes that change processor during their execution.

To serve memory operations, heaps are divided in superblocks; each superblock is formed by a set of blocks (all the blocks have the same dimension) and each block maintains a reference to its superblock. Blocks are actually used to assign memory to the processes who have requested it.

When a process p requires memory, it locks the local heap H and it checks if there are superblocks with free memory; if it is the case, a block is returned and p unlocks the heap; if there is not memory, the process also locks the global heap; if there is memory in the global heap, the respective superblocks is assigned to the heap H and a block is returned (unlocking both the global heap and the local heap); if there is not memory in the global heap, p allocates a new superblock and assigns it to the local heap H.

To free a block the process p has to lock both the respective superblock and the heap H and it has to flag the block as "free".

Allocations that are greater than a certain threshold are served directly by the operating system.

This algorithm actually does not scale: in high parallel environment, the shared heaps, which access are serialized, become a bottleneck.

Glibc is the allocator contained in the GNU C standard library; it is the standard Linux allocator. Like PTMalloc, it is derived from the Doug Lea's malloc.

This allocator is based on the existence of a certain number of heaps, called arena.

When an application gets started, by default, a single arena gets created; if, application's threads starts to suffer of too much contention on the arena; the allocator set up another arena which gets linked in a list with the older arena. This operation could be done until the number of the arenas reaches a limit that can be tunable by the user with the function *mallopt()*.

Application maintain a pre-reserved list, the *fastbin list*, of free blocks of a certain, fixed, size. This is actually a simple FIFO list: for these chunks of memory the only possible operations are take memory and release it; coalescing and splitting are not permitted. Due to the simple management of a FIFO list, if a thread needs memory of this size and that list is not empty; allocation and free could be done by exploiting an atomic operation. This way of working is known in literature as "caching of free memory blocks".

However, in general, requiring memory implies the usage of an arena, in which the access is protected by a mutex; so that accesses are serialized and only one process at a time can access this structure. The presence of multiple arenas, permit parallel allocation of memory. In the case where a thread does not find an available arena it can either wait for it or can create a new arena.

When is performed a free operation of a chunk of memory, if there is a free block contiguous to it, they can be merged independently of their size.

Very large memory allocations are managed by the system, without using heaps.

This allocator actually offers good performances but, like in the already discusses hoard allocator, when it is used in an environment characterized by a high level of parallelism, performance goes down due to the contention in the utilization of the arenas.

PTMalloc, which was the standard linux allocator for a long time, is similar to the aforementioned actual glibc allocator; in particular, it uses:

- for request smaller than 64 bytes a fastbin, thread-local list is used;
- for request between 64 bytes and 128 kilobytes, the actual arena is used in a blocking manner;
- for greater allocation, the request is handled by the actual system.

This allocator will be used later in order to compare the performances of this work. This allocator was chosen because it is known as an efficient one and its caching characteristic fits well with the designed tests.

Tcmalloc is an allocator written by Google. Tcmalloc has been written taking in mind parallel environment and so it offers good performances in those situations.

In Tcmalloc we have a heap, accessed by a central allocator, shared by all the processes; moreover, each thread has a private cache of free blocks with which small allocations are served (this technique is well known and it is called pre-reserving). Blocks from the cache are periodically taken and returned to the central heap as needed. Thanks to the private cache, small allocations can be done without the use of any synchronization primitives.

If the cache does not have free blocks to satisfy a request, a bunch of free blocks is taken from a global free-list (using a lock); if this list is also empty; new blocks are obtained from the central allocator (using another lock). For allocations greater than 32KB Tcmalloc directly uses the central, shared, allocator.

Tcmalloc does not return memory to the system. This could be a problem if it is running on a machine that runs for a lot of time before the shutdown.

Modern lock-free memory allocator

As seen in the previous section, complex and sophisticated memory management algorithms can suffer of poor performances due to the presence of critical sections that have to be executed with lock; for this reason, new lock-free memory management algorithm were born.

In this section are presented two lock-free memory allocators: LFMalloc [16] and a lock-free dynamic memory allocator presented in [42].

LFMalloc is a memory allocator that executes a lot of its operations in a lock-free manner. LFMalloc is a quite complicated algorithm: it implements what the author calls Restartable Critical Sections (RCS).

The code that is part of an RCS is executed in a lock-free manner and represents a critical section; due to this, if the thread that is executing that piece of code goes in a sleep state due to a scheduler choice, operations that it is trying to do could be wrong when it returns running; to maintain safety, if a process goes in the sleep state during an RCS execution, a notification system at kernel level will abort and restart the RCS when the process will return in a running state.

LFMalloc is, as defined by the author, mostly lock-free and it is inspired by Hoard. In particular, in LFMalloc we have the concept of superblocks, global and local heaps; and, like Hoard, if an allocation is greater than a certain threshold, it is executed directly, without using the algorithm structures. Principal differences are:

- in Hoard, we have that a local heap is binded to a thread; in LFMalloc, instead, local heaps are binded to CPU. In particular, differently from hoard, the hash function used to select the local heap takes in input the processor ID;
- because of the fact that a heap is binded to the CPU, we have no reason to have 2P heaps. In fact, the maximum degree of parallelism is P, where P is the number of CPU. So in LFMalloc we have P local heaps;
- in Hoard we have no limitation about superblocks of the same dimension; in LFMalloc each local heap can only have a single superblock of a certain size. This is not true for the global heap; it can have an arbitrary number of superblocks for each block dimension (like Hoard);
- in LFMalloc, for each processor, there are two disjoint free lists: a local one (which exploit the RCS mechanism) and a global one, which is protected by a lock. A thread will always allocate from the local list, but it might do a free on the remote list (if the blocks does not come from the local heap).

The allocation tries to remove, from the local heap, a block from the superblock which contains blocks of the right size exploiting RCS; if it is successful, the block is returned; else a new superblock of the required dimension will be taken from the global heap and it will replace the current superblock (which does not have any free block); the allocation is tried again on this new superblock. Notice that the fact that this superblock replaces the old, empty one is done in order to respect the hypothesis that every local heap has a single superblock for a certain size.

Free operation could be done in two different manner depending on the heap status. If the block being freed belongs to a superblock which is still present in the local heap, the operation is done in a lock-free manner, exploiting RCS. If, instead, the block belongs to a heap which is no more local, free is done exploiting classical blocking algorithm.

Michael proposal in [42] exploits the concepts of superblock, multiple heap and free list; it only uses atomic operations (like CAS and LL/SC) obtaining a dynamic memory allocator which is completely lock-free.

Modern wait-free memory allocator

The wait-free memory allocator described in [29] is a buddy system memory allocator. The idea behind this allocator consists in building a binary tree where each node represents a chunk of memory and maintains, inside it, its dimension. The two child of a node refers to the same memory, divided in two buddies. Allocation is possible exploiting an atomic operation of decrease: the allocator, recursively (starting from the root and continuing in the sub trees), tries to subtract in the node the number of bytes that it is trying to allocate. Recursion terminates when:

- the subtraction returns a negative value: in this case there is not sufficient memory;
- the subtraction returns the value 0: in this case we have reached a node of dimension exactly equal to what we need. This node represents the chunk of memory that we will deliver. Sons of this node will have an undefined value.

The free of a block will re-initialize the value of the sons and, will atomically add, in each ancestor of the freed node, the size of the byte just freed.

This algorithm, in my opinion raises a problem not discussed by the author. Suppose that, the root, contains a value of 5. This value is obtained by 4 bytes, free in the extreme left of the tree and 1 byte, free in the extreme right. If we try to allocate 5 bytes, it will be returned the root. However, this memory is not contiguous.

1.4 NUMA architecture

NUMA architecture was born in the context of highly parallel machines composed by a lot of CPUs, to limit the number of cores serialized to access the central memory. In a NUMA machine a certain bank of memory can only be accessed by a core (or a restricted number of cores) with a private BUS. The existence of this BUS, which can be accessed by a little number of cores, achieves the purpose of reducing memory

access contention (and so reducing memory access latency). Banks that a certain core can access with the private BUS are called "private memory of the node". The union of cores and their private memory define a "NUMA node".

Obviously, there is the possibility, for a certain core, to access memory which is on another node (this is called "remote memory" for the core). This is possible by the interconnections of the various nodes, done by using high performance BUSes; any interconnection topology is possible but obviously a fully meshed network gives better performances in most cases (an example of this type of interconnection is shown in figure 1.4, where the blue lines represent the shared BUSes).

If, during the execution, a process P, which is running on a certain NUMA node N, require a data stored on an another NUMA node N', a message-passing coherence protocol is used. This allows P to access remote memory but it actually reduces performances for the following reason:

- a copy of data has to be done (this is actually pure overhead);
- the message passing protocol that has to be used in order to exchange data, invalidate cache lines;
- the BUS used to exchange data and messages could be a shared bus: in this case cores that need remote data have to wait for the BUS availability;
- maintaining coherences between informations present in multiple NUMA nodes could be very difficult and heavy.

From what said above it easily follows that the efficiency of a NUMA machine is strongly in function of how many access to the remote memory are done.

In order to increase performances NUMA policy were born; a NUMA policy tells us in which node the thread have to allocate memory. They are:

- default: the process allocates in the node where it is running, this policy is also known as "first-touch policy";
- bind: the process tries to allocate memory in a specific set of nodes; if memory is not available in that nodes, allocation fails;
- preferred: it is similar to the bind policy, however, in this case, if memory is not available in the specified nodes, memory is allocated in another node;
- interleave: it alternates the allocation in a set of nodes.

NUMA architecture need to be supported by the operating system which is required to offer facilities to drive memory management. Linux is NUMA-aware

starting from kernel version 2.6. It offers a set of facilities to drive memory allocation (e.g. allocate in a certain node or on a set of nodes), to impose a process to run on a specified node, to move from a node to another one.

For simplicity, and due to the fact that it is difficult to predict what a thread is doing, Linux kernel does not perform any page migration from a node to another one; if a process changes node due to a scheduler choice, pages migration has to be done by the program itself. Moreover, in absence of policy chosen by the programmer, kernel simply allocates memory exploiting the first touch policy.

To simplify programming code that exploits NUMA architectures, a user level library (numa.h) [30] has been developed. This API gives simple functions to call in order to migrate pages, bind processes on a specific node, choose the NUMA policy and allocate memory in a particular node.

An allocator should know what are the peculiarities of this kind of architecture in order to maximize performance, reducing as much as possible the need to access remote memory.

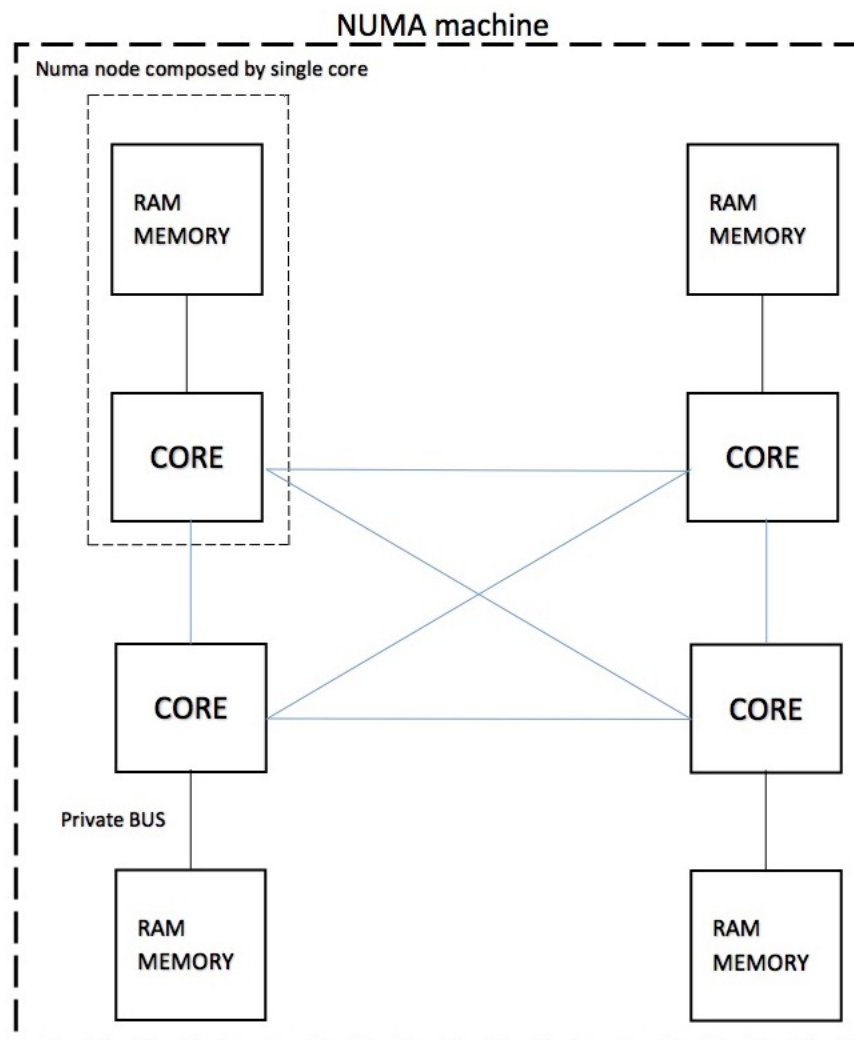

Figure 1.4. Example of a NUMA machine with a meshed topology

Chapter 2

Lock-free allocator: a brief introduction

In this work is presented the conception and the implementation of a lock-free buddy system.

The algorithm is actually very different from classical buddy system implementations (a pseudo code can be found in Algorithm 2 and Algorithm 3). In classical implementations we have the following characteristic:

- there is a certain number of free lists, with each of this list containing blocks of a certain size (the size can only be a power of two);
- at startup, we have a single block of dimension equal to the maximum allocable size managed by the buddy system;
- when an allocation requires a block of size s: if there is not a block of this size, bigger blocks are divided until reaching a block of size equal to the nearest power of two of s; blocks produced during this process and not used are inserted in the opportune lists;
- when there are two adjacent free blocks of the same dimension s, these are merged together in order to build a block of size 2s;
- free list management uses locking algorithms.

We can observe that free list management represents an intensive operation and it can cause performance problems.

In this work I present an alternative implementation of a buddy system memory allocator which does not use free lists. In this proposal, the buddy system is realized with a balanced binary tree and we can highlight these major characteristics:

Algorithm 2 Classical buddy system allocation

```
 1: procedure ALLOC(int size) return void*
 2:     size = round_power_two(size)
 3:     if ∃ a block in the free list of size bytes then
 4:         return the top block of the free list
 5:     current_list = the list with blocks of size equal to "size"
 6:     while current_list is empty do
 7:         current_list = the list with blocks of size equal to current_list.size*2
 8:     if all the lists are empty then
 9:         return NULL
10:     b = the top block of current_list
11:     while b.size ≠ size do
12:         split the block
13:         b = the first splitted block
14:         put the second block in the right free list
        return b
```

Algorithm 3 Classical buddy system free

```
1: procedure FREE(void* block)
2:     while b has an adjacent free block b' of size equal to b.size do
3:         remove b' from its free list
4:         b = merge(b, b')
5:     put b in the opportune free list
```

- the root of the tree represents the entire chunk of memory that the buddy is going to manage;
- each level logically represents a free nodes list of a certain size;
- if nodes in level i have a size s, nodes at level (i+1) have size ($s/2$); moreover, children of a node represent the same portion of memory, divided in two smaller blocks of equal size;
- the tree is a static structure; i.e. it is initialized at startup and it does not change[1]. The algorithm will be concept in a manner such that splitting and coalescing operations will exploit the structure: so, they will not cause any overhead;
- if a process, at a time t requires a certain chunk of memory m and, at a time $t' > t$ it needs more memory, contiguous to m, due to the tree organization it can easily know if it possible. This is actually a new feature for this kind of memory management: in classical buddy system, this operation can only be done by a linear search on the free lists;
- the algorithm is lock-free.

[1]The solution can be easily extended to allow it to be possible that to dynamically allocate a new buddy, if it become necessary for memory reasons

An example of the basic data structure can be found in Figure 2.1. In this simple example, we have a binary tree capable to manage 4 bytes of memory. In each node, we have the dimension of the managed memory and its offset respect to the start of the managed memory (in the field `start`). We can see that, the left child starts at the same offset of the parent; the right child starts at the half of the memory managed by the parent: this highlight the fact that children are in the same portion of memory of the parent.

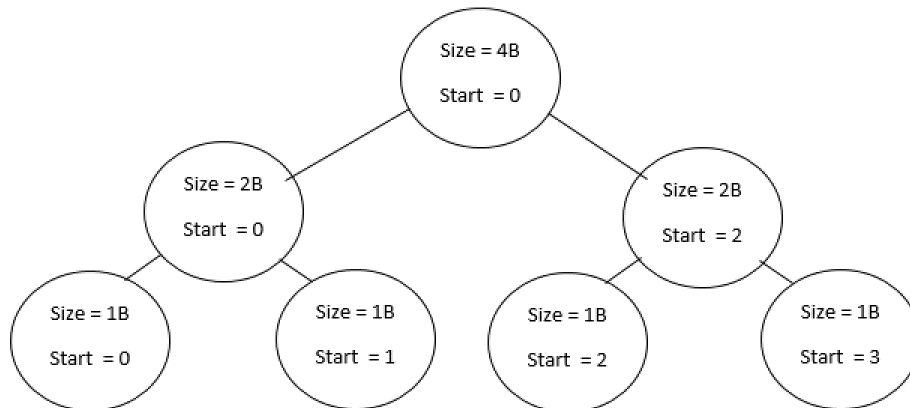

Figure 2.1. Basic representation of the data structure

Classical buddy systems have a number of free lists which is limited due to the splitting and coalescing costs; for example, linux buddy system manages eleven free lists, offering the possibility to allocate from one page (4KB) to 10 pages (4MB, which is the dimension of a linux "huge page"). With this proposal, due the zero cost of splitting and coalescing, theoretically, we have no limits on the number of logically free lists that it is able to manage; it could be possible, simply tuning a parameter, to permit the allocation from one page to all the possible allocable memory. However, this feature has to be used with care: the buddy system permits the management of fixed size memory, if we make possible to allocate large portion of memory (relatively to the size of the entire managed memory), it might happen that no other allocation could be possible. In classical utilizations of this proposal, it will be possible to allocate memory from leaves to an arbitrary higher level, that can be different from the root.

2.1 NUMA integration

NUMA integration will be done by replicating data structures needed by the algorithm in each NUMA node.

When a process has the necessity to allocate memory; it will start searching

in the tree dedicated to its NUMA node; moreover, to maximize the number of parallel processes, each one will try to search for a free chunk starting by a different sub-portion of the whole tree which is identified relying on a hash function of its process id.

If the allocation in the tree relative to the current NUMA node fails, the process could start to search for a free chunk in other NUMA nodes. This possibility has to be spelled out by the calling process; in fact, it could be possible, for a process, to desire only local memory; this kind of behavior could be specified simply by passing a flag to the allocation function.

Each process will run an instance of the memory allocator that will maintain data structures for all the NUMA nodes. The fact that all the nodes are easily accessible can be useful in some cases:

- a certain thread, during the execution, changes node but it has to free memory in the old node;
- as previously said, with this proposal, it is possible to allocate memory in a specific address in order to take memory contiguous to previously allocated address (if it is free): with this organization, it is still possible in the case in which a thread changes node;
- this organization simplify the management in all the cases in which the selected NUMA allocation policy implies to control a set of nodes;
- in some application, we have data scattering: there is a thread that allocates all the memory needed and it distributes it to all the threads of the application. The organization of this proposal does not need to utilize different buddy system instances to allocate memory for thread running in other nodes.

Chapter 3

The algorithm and the implementation

In this chapter is presented and discussed the actual algorithms. better understand the mechanisms that regulate its operation, it is introduced the first, simpler, version and then it is introduced the final one presenting the differences between the two implementations. In this manner it is simpler to:

- highlight which are the principal steps that led to the creation of a quite complex algorithms;
- have an easier and better understanding of the algorithms;
- present properties and peculiarities are easier to be discussed on the first version and naturally extended to the final algorithm.

The basic idea of the algorithm is introduced in section 3.1. With this version of the algorithm considerations relative to the ABA problem and the lock freedom will be done. After that, in section 3.2 a newer, better version of the algorithm (with the same identical logic of the first one) is presented. In section 3.3 is provided a reviewed version of the algorithms, which is slightly different and, still implementing the same logic, actually refined. Finally, shortly, the NUMA implementation of the last version of the algorithm is discussed in section 3.4.

3.1 The idea

The basic idea is composed by a binary tree, where each node of the tree is associated to a segment of memory. Each child of a node represents a portion (an half) of the parent's chunk of memory, the root represents the overall allocable memory of the buddy system and the leaves represents the minimum allocable memory.

The memory is actually a big pre-allocated segment and each node refer to it using an offset, representing the distance, in bytes from the start of the segment.

The structure that represents the node is as follows:

```
typedef struct _node{
        unsigned long mem_start;
        unsigned long mem_size;
        unsigned long val;
        unsigned pos;
} node;
```

Structure 3.1. node's structure

The structure contains the following informations:

1. `mem_start` represents the offset of memory chunk associated to the node, relatively the overall memory space;
2. `mem_size` is the size of the memory associated to the node;
3. in the field `val` we found informations needed to represent the state of each single node;
4. `pos` is the node's position in the binary tree;

Each node can be in one of the following state (or in a combination of them):

- occupied: this state represents the fact that the memory represented by the node was fully delivered to some calling process. In this state, the descendants of the node are not allocable (the memory is already occupied) and, the ancestors of the node are in the state "partially occupied". This state must be used only in combination of the states partially occupied left and partially occupied right;
- partially occupied: if a node n is partially occupied, it means that a certain node (or a set of them) in the subtree rooted in n is occupied. We can recognize three different types of partially occupied states. We have the states: "left" if the occupied subtree is on the left child of n; "right" if it is on the right subtree; "left and right" if both the subtrees are occupied. In those states the node is not allocable;

- coalescing: this state is the successive state of a "partially occupied" state. It means that the process who has set this node as partially occupied (in a certain side) is going to free it. Due to the fact that the node is not freed yet, a node that is coalescing can not be fully allocated. Coalescing information is actually necessary to cope with concurrent operation on the same node. Process performing a free operation has to be informed if other operations were done on that node and so it must not do any further modifications (as we will see);
- free: a node is free if no one process has occupied (partially or totally) it. A node which is free can be allocated both totally and partially if requested.

It is important to notice that, for performance reason, when a node is successfully allocated, descendant nodes (which are no more allocable) are not touched and they appear as free nodes. This situation does not compromise the correctness because processes follows the rule: *bigger allocations have priority; if an allocation of a node n, reaches a node n', ancestor of n, which is occupied, the allocation has to fail*; however, if a process tries to take a node and after it discovers that an ancestor is occupied, it has to abort the allocation and rollback the state of the nodes that it has wrongly modified.

To make it possible to change the state of a node, without any transient and without using any blocking technique, we rely on atomic instructions. As seen in the first part of this work, this kind of operations could be used only to manage single word. This implies that it is necessary to use a single word that is capable to represent all the possible node's states. To do this we will use the last five bits of an `unsigned long` and we will represent states by using bitwise operations. The `unsigned long` that will represents meta-information of a single node, is organized in the following manner [Figure 3.1]:

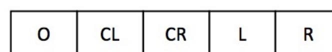

Figure 3.1. Node's possible state

From left to right it is possible to recognize the five states/fields: **O**ccupied, **C**oalesce **L**eft, **C**oalesce **R**ight, **L**eft occupied, **R**ight occupied. For convention, when a node is occupied, also the L and R bits are set. Moreover, for reasons linked to the algorithm behavior, if the coalesce bit of a side is set, also the occupied bit of that side has to be set.

In the presentation of the algorithms, to augment the code readability, I will use

the following MACRO [1]:

```
OCCUPY = 0x10

NOT_OCCUPIED = (~(OCCUPY))

CLEAN_LEFT_COALESCE = (~(LEFT_COALESCE))

CLEAN_RIGHT_COALESCE = (~(RIGHT_COALESCE))

OCCUPY_LEFT = 0x2

OCCUPY_RIGHT = 0x1

LEFT_COALESCE = 0x8

RIGHT_COALESCE = 0x4

CLEAN_OCCUPIED_LEFT  = (~(OCCUPY_LEFT))

CLEAN_OCCUPIED_RIGHT = (~(OCCUPY_RIGHT))
```

Code 3.2. MACRO used in the algorithms

3.1.1 Allocation

The allocation is formed by two function, *alloc()* and *check_parent()*. Moreover, the *free()* function could be called in case of allocation failures.

The first algorithm presented is *alloc()* Algorithm 4. The algorithm takes in input a node n, which is supposed to be free, and it starts the allocation process by trying to assign to the node the correct value. This write is done by a Compare And Swap (CAS) and it may fail; in particular it fails if, concurrently, some other thread occupies, partially or totally, the node n.

If the CAS does not fail, the algorithm has two possibilities: the first one is that the written node is the root and, in this case (since descendants are not touched),

[1] in the MACRO, **0x** indicates a hexadecimal number and ∼ represents the bitwise NOT operation

the allocation is completed and the function returns true; the second possibility (the general one) is that n is not the root: in this case, to complete the allocation, a call to the function *check_parent()* Algorithm 5 is needed. In real utilization of the allocator it will be hardly permitted to allocate the root, because, as mentioned above, it would mean to deny other allocations in the buddy system.

The function *check_parent()*, which will be presented soon, tries to set the ancestor nodes as "partially occupied". *Alloc()*, when completed, return *check_parent()*'s result and if it has returned true, the node n is considered definitely occupied; otherwise it means that an ancestor node is discovered as fully occupied and the allocation is failed for the aforementioned priority rule.

Algorithm 4 Allocation simplified

1: **procedure** ALLOC(node* n) **return** bool
2: $preso \leftarrow OCCUPY \mid OCCUPY_LEFT \mid OCCUPY_RIGHT$
3: $trying \leftarrow n$
4: $actual \leftarrow *n$
5: **if** $actual \neq 0$ **then**
6: **return** false
7: **if** $!CAS(n, actual, preso)$ **then**
8: **return** false
9: **if** n is the the root OR check_parent(n) **then**
10: **return** true
11: **else**
12: **return** false

check_parent() function takes in input a node n, already marked as occupied (totally or partially) and it tries to mark its parent as partially occupied; in particular, it tries to mark the bit relative to the subtree where n is present. Cases to take in account are the following:

- parent(n) is free: this is the simplest case in which the procedure only tries to set the bit as it need;
- parent(n) is occupied: this is the case where the fifth bit of the parent is equal to 1. In this situation the operation will fail and it is needed to restore a coherent state (as explained below);
- the subtree in which n resides is already occupied. This situation actually is not a problem, in fact, it simply means that a brother of n (or a descendant of n's brother) has been already occupied. Due to the fact that it is not a problem, the algorithm simply sets again the bit as occupied and cleans the coalesce bit (to avoid that concurrent free operation cleans the occupied bit);
- the subtree in which n resides is in a coalescing state. In this case, the *check_parent()* algorithm cleans the coalescing bit and sets again the "partially

occupied" bit, the free operation that has set the coalescing bit will halt in this node because of the fact that its bit has been already cleaned.

The fact that the core of the algorithm is in a do - while cycle that implements an abort-retry mechanism, is typical of non-blocking algorithms and it is done since concurrently, the state of the node might change due to other nodes operations. In particular, it might happen that:

- some modifications on the other subtree (this is not a problem at all);
- the coalescing bit of the subtree where the node *n* is, becomes 1. This happens due to a free operation working on the brother (or on a descendant of the brother) of *n*. In this case there is the possibility to clean soon the coalescing bit;
- the partially occupied bit relative to the subtree of *n* becomes 1 by the hand of a brother (or a descendant of the brother) of *n* and this is not a problem at all;
- the node becomes totally occupied: in this case, for the priority rule, our allocation has to fail. In case of failure the procedure cancels the modifications already done calling the free function. The free function will work from the node trying (which is the node that the allocation tried to take) to the *upper_bound* node which is the node where the allocation is failed.

Algorithm 5 Check parent simplified

```
 1: procedure CHECK_PARENT(node* n) return bool
 2:     repeat
 3:         actual_value ← *parent(n)
 4:         if actual_value & OCCUPY ≠ 0 then
 5:             free(trying,n)
 6:             return false
 7:         new_value ← actual_value
 8:         if n è sinistro then
 9:             new_value ← new_value & CLEAN_LEFT_COALESCE
10:             new_value ← new_value | OCCUPY_LEFT
11:         else
12:             new_value ← new_value & CLEAN_RIGHT_COALESCE
13:             new_value ← new_value | OCCUPY_RIGHT
14:     until !CAS(parent(n), actual_value, new_value)
15:     if parent(n) is the root then
16:         return true
        return check_parent(parent(n))
```

As a final note of the allocation process, aiming to reduce contention, the starting node passed to the *alloc()* function is obtained considering the process ID (PID) of

the thread that is requesting memory, in order to avoid that too much processes are trying to allocate the same chunk. In particular, if a process P requires X pages and is defined LS as the last node of size X, FN as the first node of size X and NOP the total number of the processes that the allocator is serving, the starting node where the function tries to allocate memory is computed by:

$$starting_node = (PID\%NOP)\%(LS - FN).$$

Moreover, if an allocation fails on the ancestor node N, the next attempt will be on the first node which is not in N's subtree, since it is fully occupied by the allocation in N.

3.1.2 Free

The free is composed by two functions, *free()* and *unmark()*.

free() takes in input the node n, which is the first node to be freed and the *upper_bound*, which is the last node to be freed. The *upper_bound* can be: the root if n is a node that has been really occupied; any node if the free is called by a failed allocation. The procedure is composed by three steps:

1. coalescing bit of the ancestors (until the *upper_bound*) of n are marked [lines 5-13]. These bits will be used by the *unmark()*) function to understand if its work is still necessary;
2. n is freed by setting its value to 0 [line 14];
3. coalescing and partially occupied bits (until the ***upper_bound***) relative to the subtree where n resides are cleaned (this work is done by the function *unmark()*).

unmark(), which is presented in Algorithm 7, is a recursive function that takes in input a node n, which is the son of the node n that the function has to manipulate, and a node ***upper_bound*** which is the last node to clean; the function does not return any value.

The function starts by checking if the coalescing bit in *n'*, relative to the subtree where n is present, is still set: if it is not the case, it returns, breaking the recursion. This scenario is possible if a new concurrent allocation or a concurrent free operation of the brother (or a descendant of the brother) has already reached *n'*. If the bit is still set, the function tries to clean both the coalescing bit and the partially occupied bit relative to the subtree where n is present. This operation, as usually, is done with a retry mechanism to avoid possible problem with concurrent operations. Once the operation is correctly done, the function does the following checks:

Algorithm 6 Free simplified

```
1: procedure FREE(node* n, node* upper_bound)
2:     node* actual ← parent(n)
3:     node* runner ← n
4:     while runner ≠ upper_bound do
5:         repeat
6:             actual_value ← *actual
7:             if left_child(actual) = runner then
8:                 new_value ← actual_value | LEFT_COALESCE
9:             else
10:                new_value ← actual_value | RIGHT_COALESCE
11:        until (!CAS(actual,actual_value, new_value))
12:        runner ← actual
13:        actual ← parent(actual)
14:    *n ← 0
15:    if n ≠ upper_bound then
16:        unmark(n, upper_bound)
```

1. if *n'* is the *upper_bound*, the function breaks the recursion and returns;
2. in the case where *n'* has the occupation bit relative to the brother of n set to 1, the function breaks the recursion. This happens because, in this case, continuing to clean the ancestors would be an error: in fact, the subtree is used by the brother of n. This situation is depicted in Figure 3.2, where is it possible to see that, if the red bit is clean, informations about the occupied node on the extreme left would be lost.

In the cases where the above controls are evaluated as false, the recursion continues.

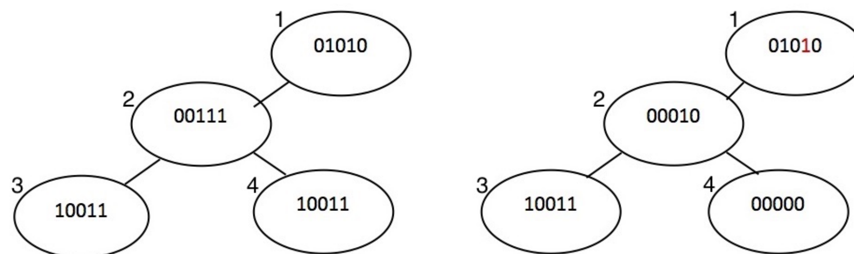

Figure 3.2. Free of the node 4, occupied and coalescing bits remains marked in the node number 1

It is important to do the following observations:

- in the case in which the recursion terminates due to the second control, ancestor's coalescing bit will remain marked; nobody will clean it until a new allocation or a new free will reach that node. This implies that we can not be sure that a node with both the coalescing bits marked is actually free. For this reason, we can only allocate nodes with value equal to zero;

- in the case of two concurrent free operations (of n and its brother), only one of the two instances of the function *unmark()* will be admitted to continue the recursion, it is ensured by the fact that:

  1. at line 14, one of the two CAS will fail because it will find the brother's bit with value 0;
  2. the first process that terminates the CAS will have the brother's occupation bit marked and so it will terminate the recursion at lines 16-17;
  3. the process that failed the CAS, will see the brother's occupation bit cleaned and so it will continue the recursion.

Algorithm 7 Unmark simplified

```
 1: procedure UNMARK(node* n, node* upper_bound)
 2:     actual=parent(n)
 3:     repeat
 4:         actual_value ← *actual
 5:         new_val ← actual_value
 6:         if n è sx AND ((actual_value & LEFT_COALESCE)=0) then
 7:             return
 8:         else if n è dx AND ((actual_value & RIGHT_COALESCE)=0) then
 9:             return
10:         if n è sinistro then
11:             new_val ← new_val & CLEAN_LEFT_COALESCE
12:             new_val ← new_val & CLEAN_OCCUPIED_LEFT
13:         else
14:             new_val ← new_val & CLEAN_RIGHT_COALESCE
15:             new_val ← new_ val & CLEAN_OCCUPIED_RIGHT
16:     until !CAS(actual,actual_value,new_val)
17:     if actual = upper_bound then
18:         return
19:     if n è sinistro AND ((actual_value & OCCUPY_RIGHT) ≠0) then
20:         return
21:     else if n è destro AND ((actual_value & OCCUPY_LEFT) ≠0) then
22:         return
23:     else
24:         unmark(parent(n), upper_bound)
```

As a final note, I want to highlight the fact that these algorithms do not suffer of the ABA problem. In fact, we observe that the ABA could happen only after a successful Compare And Swap. In this particular application possible scenarios are:

- an allocation or a free operation tries to perform a CAS and it fails. In this case it is impossible to have ABA problem;
- p tries to set a node as "totally occupied" and the CAS succeed. It means that the value of the node is equal to what the process has seen at the beginning of the allocation process (and, in particular, due to the fact that the allocation

is continuing, it means that the node has the value equal to 0): if this value is actually the one seen by the process, obviously there are no problems; if the actual value is an updated value, i.e. we are in a scenario in which *p* was descheduled and, in the meanwhile, another process, *p'* has occupied (partially or totally) the node and, it released it before the *p* reschedule, actually we do not care. For this kind of applications, it is important that, in the moment in which a process executes the CAS, the node is free; it does not matter if the node was previously allocated;

- in the first step of a free operation, where parent nodes are marked as "coalescing", it is not possible to suffer ABA problem due to the fact that, the algorithm is simply setting the right bit. It can be sure that the operation is correct due to the fact that we are coming from a node that was successful occupied before;
- in the second step, when the value of the occupied node is set to 0, we obviously can not have ABA problem. The node is still occupied by the current operation and so no other process is able to update it;
- in the end, when we clean the marked parents, it can actually happen that another process, which is freeing a brother of the node *n* on which we are working, had set again the coalescing bit. However, this does not matter. One of the two processes have to clean the coalescing and the occupied bits, and the other one has to stop its operation because its work has been already done. This ABA does not compromise correctness at all.

3.1.3 Lock freedom proof

To demonstrate lock-freedom we have to take each retry cycle and check if it can fail forever, with no process making progress; lock-freedom requires that, if a retry cycle fails, it means that there is another process which is making progress, moreover, this property admit starvation, i.e. it is acceptable a schedule in which a process is never able to make progress, while other ones goes ahead.

We notice that logically a *free()* makes progress, i.e. it ends, when it is able to actually free its block. Differently, *alloc()* makes progress if:

- it is able to allocate a new block;
- the allocation fails, i.e. it discovers that the subtree, where the node *n* that it is trying to reserve, has an occupied node *n'* which is an ancestor of *n*. In fact, remembering the priority rule explained before, we have that, if an allocation found an occupied ancestor, the allocation has to fail.

I will now examine each single function. The *alloc()* procedure, presented in Algorithm 4 does not have any retry cycles, the part of the allocation that needs to be discussed is the part done by the auxiliary function, *check_parent()*, which is called by *alloc()* and it is presented in Algorithm 5. To understand if the function is lock free we have to check lines [2-14], which are part of a retry loop. Possible situations are:

- the check at line 4 fails, i.e., we are in the case in which the node it is trying to mark is fully occupied, the function has actually made progress: it has reached a final state in which it is able to signal the impossibility to allocate a node in that subtree;
- the Compare And Swap operation is successful: in this case the function is actually making progress;
- the Compare And Swap operation is not successful: this happens if someone else has modified the node during the execution of the procedure. This can happen for multiple reasons:
    1. a free operation has cleaned the coalescing bit and it is going on the upper node to continue, or another *check_parent()* instance has set a partially occupied bit and it is already making progress. In this situation, where two or more function are interested to flip a single bit of the same instance of the node (i.e. they see the same value of the node), it is possible to observe that one function is able to complete the CAS operation. This function will continue its work on the upper nodes (making progress) while other functions will read the update node's value and then will retry. This situation respects lock-freedom property;
    2. in the meanwhile, an allocation has fully occupied the node. In this case the operation will fail in the next iteration of the loop (if this node is still busy when it is retying) falling back in the aforementioned case.

free() function is divided in two steps.

In the first step, when the function marks as coalescing the ancestors of the node n being freed, there is a retry loop in the lines [5-11]. This portion of code can not fail: it retries until it is not able to mark the node as coalescing. The Compare And Swap could fail a certain number of times for the following reasons:

- a *check_parent()* function cleans the already marked coalescing bit (this could happen in both subtrees) or it marks the bit relative to the n's brother subtree. In this situations, *check_parent()* is making progress and so the lock freedom

property is satisfied. I want to highlight the fact that, if *check_parent()* simply marks again the partially occupied bit relative to the n's subtree, both the *free()* function and the *check_parent()* will make progress;

- another *free()* marks/cleans one of the two coalescing bit: in this case the latter is making progress.

The second step of the *free()* is done by *unmark()* function. This procedure breaks the retry loop when it is able to complete the CAS operation or when it finds out that the work it has to do has been already done by someone else. The motivation in which we need to iterate on the loop are multiple:

- there is a modification on the other subtree (allocation, or one of the steps of the free function). In this case the other operation is making progress;
- another *free()*, or an instance of *check_parent()*, which is acting on the same subtree in which we are working on, clean the coalescing bit: in this case it makes progress going on the upper node and we also make progress because in the next iteration we will break the recursion.

I want to highlight the fact that, it could not happen that, another free operation, marks again the coalescing bit relative to the same subtree in which we are working on, leading to a situation in which we do not see this modification (ABA). In fact, to make it possible, it might mean that a brother of the node we traversed is occupied and concurrently it is going to be freed. If this is the case, the *unmark()* operation would stop before the recursion step (stopping the recursion means to terminate and so make progress) thanks to last controls of the function; it is not possible also in the situation in which that brother becomes occupied in the meanwhile: in this case, it would clear the coalescing bit and the function would halt in the controls at the beginning of the retry cycle (also in these cases terminating and so, making progress).

3.2 Practical improvements

Starting from the simpler version of the code presented in the previous section it is straightforward to do some useful tuning in order to optimize algorithms performance.

The reference architecture for this work is x86-64, which implements Total Store Order (TSO) memory consistency, where, as shown in the introduction, atomic operations can cause some performance issues due to their interaction with the processor's write buffer and cache memories; moreover, other performance criticisms are related to the retry cycles which can be iterated a great number of time, until the Compare And Swap is successful.

It follows that is a good idea to try to limit atomic operations (the Compare And Swap in our case) as much as possible.

In this section is presented a reviewed version of the algorithms which uses a compare and swap every three logical operation. The logic behind the algorithms is identical to those already presented and so are explained only the organization differences from the previous one.

First of all, the struct node is slightly different:

```
typedef struct _node{
        unsigned long mem_start;
        unsigned long mem_size;
        unsigned pos;
        unsigned long* container;
        char container_pos;
} node;
```

Code 3.3. Struct node improved

This new structure is composed by the fields: *mem_start*, *mem_size* and *pos* are exactly equal to the fields of the previous version; the field *container* does the work of the *val* field of the previous version.

The big difference from the previous solution is that in the container, we have meta-information about 7 nodes (three levels of the tree) instead of a single node: in this variable, each node has its reserved 5 bits.

Because of the fact that a single word (container) is shared by seven nodes, it is needed, for each node, to know which is its position inside the container (other than its tree's position). To this end, the variable *container_pos* is inserted (this variable represents an integer number but, due to the fact that it can only be in the interval [1-7], a char has been used).

The Figure 3.3 graphically represents the organization. In this figure is possible to see that the tree is represented by an implicit representation that exploits an array; in this array each entry is an element of type *struct_node*.

In this version, each container maintains informations about a bunch of tree (where each bunch is formed by a father, 2 childes and 4 grandsons). So, it does not happen that in a container we find nodes that are all in the same level (if this would happen, we would not be able to reduce the number of CAS). All the functions are being re-implemented following the same schema:

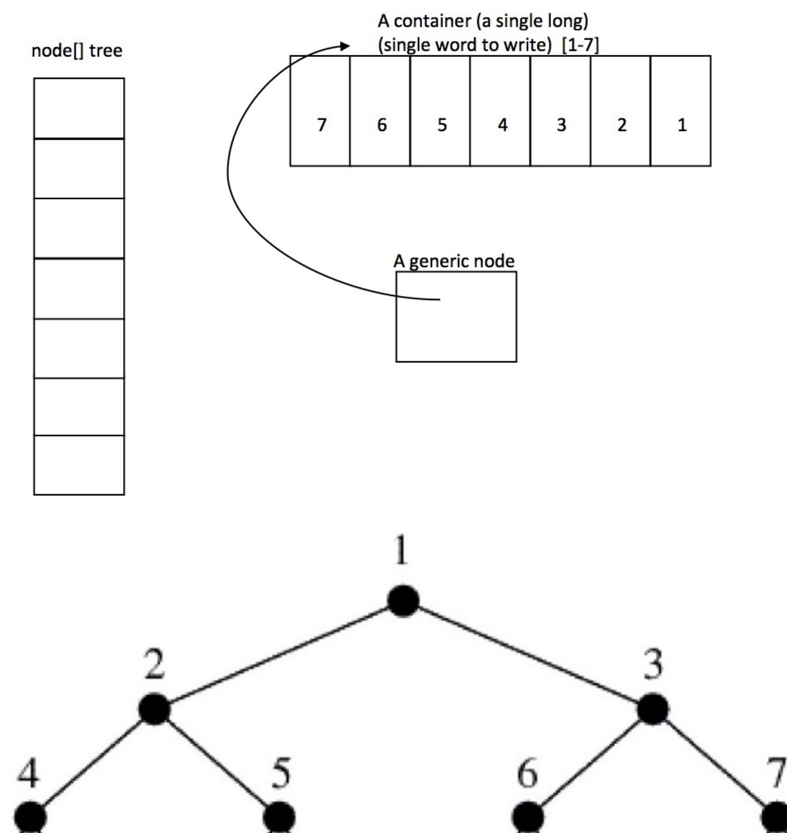

Figure 3.3. Three levels organization

1. the core of the procedure has been moved from the principal function to an auxiliary one that do operations on the single node in its container;
2. the principal procedure, when it is called to a node n, will seek to repeat the auxiliary function three times if n is a grandson (position 4,5,6,7 of the container), two times if n is a son (position 2 and 3 of the container) and one time if n is the root of its bunch (position 1 of the container): this logic is implemented simply using switch/case;
3. once the operation on the root of the bunch has been done, the principal function tries to write the updates by using the CAS on the container. If CAS fails, all the operations that have been previously done have to be done again.

Considerations relative to the ABA problem and lock freedom property easily extends from the previous version of the algorithm.

3.3 Optimized algorithm

In order to further reduce CAS, remembering that atomic operations can only work with a single word, a set of small modifications to the algorithm data structure is required. In fact, by maintaining the previous organization we would need 75 bits to add a level which is more than the dimension of a single word (64 bit in standard x86_64 machines).

In this version of the algorithm we have the following structures (a graphical representation is done in Figure 3.4):

```
typedef struct node_container_{
        unsigned long nodes;
        node* root;
}node_container;
```

Code 3.4. Struct node container

```
typedef struct _node{
        unsigned long mem_start;
        unsigned long mem_size;
        unsigned pos;
```

```
        node_container* container;
        char container_pos;
} node;
/* Tree definition: */
node* tree;
```

Code 3.5. Struct node and tree definition

We maintain the implicit representation of the binary tree that we saw in the previous section in the global variable tree.

Tree's entries are still of type *struct_node* which is actually slightly different. In particular, struct's field are the following:

- `mem_start`: this field is still an offset (relatively to the allocable memory) and tells us where the memory allocable by this node starts;
- `mem_size`: says how big is the memory allocable by this node;
- `pos`: is the position of the node inside the tree;
- `container`: represents the node state. States management is the core difference of this version of the algorithm from the previous version. Status management will be explained in detail below;
- `container_pos`: is the position of the node in the container (it can be from 1 to 15 as we will see).

The other important structure of the buddy system is `node_container`. This structure is used to atomically update 4 levels, which we will call "bunch", of the tree by a single CAS (4 levels means that there are 15 nodes); in this structure, we have a field "nodes" in which there is the state informations about the nodes that belong to this container and a pointer to the root of this bunch.

In the presentation of the algorithms the following MACRO are used:

- TREE_ROOT: it represents the root of the global tree, i.e. tree[1];
- ROOT(n): it gives back the ROOT of n's bunch;
- IS_ALLOCABLE(val, n): tells us if n is allocable (i.e. the value of the node is 0), according to val;
- LOCK_NOT_A_LEAF(val, n) and LOCK_A_LEAF(val, n): are used to mark a non-leaf node or a leaf node as occupied;

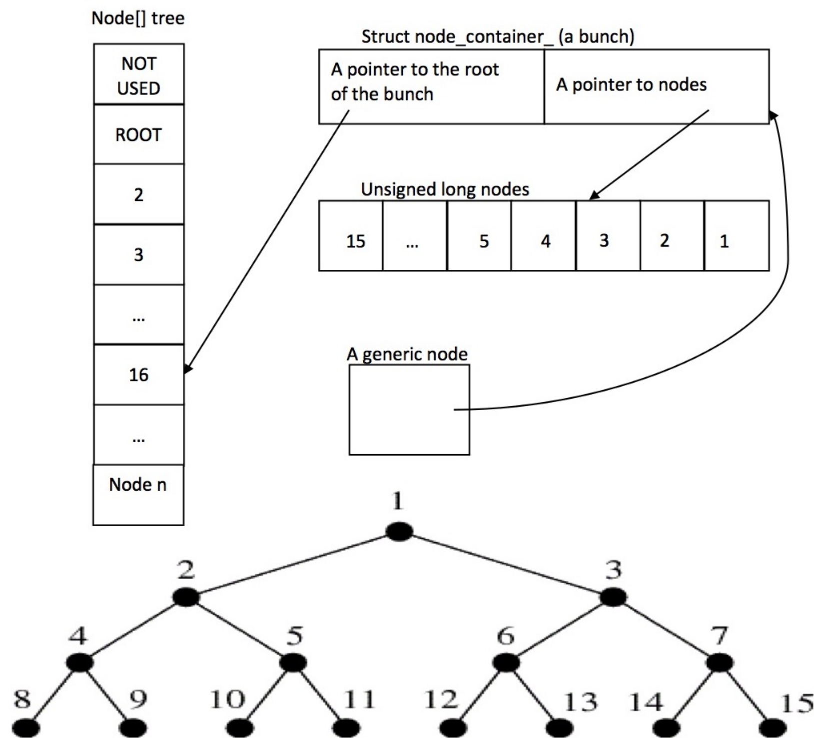

Figure 3.4. Four level organization

- IS_OCCUPIED(*val, n*): tells if a node is totally occupied (i.e. a node that is partially occupied will cause a "false" return);
- UNLOCK_NOT_A_LEAF(*val,n*) and UNLOCK_A_LEAF(*val,n*): dual of LOCK_NOT_A_LEAF(*val, n*) and LOCK_A_LEAF(*val, n*);
- COALESCE_LEFT(*val, n*) and COALESCE_RIGHT(*val, n*): are used to mark n as coalescing (left or right respectively);
- CLEAN_LEFT_COALESCE(*val, n*) and CLEAN_RIGHT_COALESCE(*val, n*): duals of COALESCE_LEFT and COALESCE_RIGHT;
- IS_OCCUPIED_LEFT(*v, n*) and IS_OCCUPIED_RIGHT(*v, n*): tells us if *n* is partially occupied (left or right respectively);
- parent(*n*), left(*n*), right(*n*): are used to explore the tree.

To make it possible to put states of fifteen levels in only a word, differently from the previous version of the algorithm, here only the leaf of each bunch maintains the five state bits, nodes on the other levels only maintains a bit of information that represents the status occupied (with no differences for partial or total occupation) or free. Other informations, like partially occupied, or "in coalescing" could be inferred

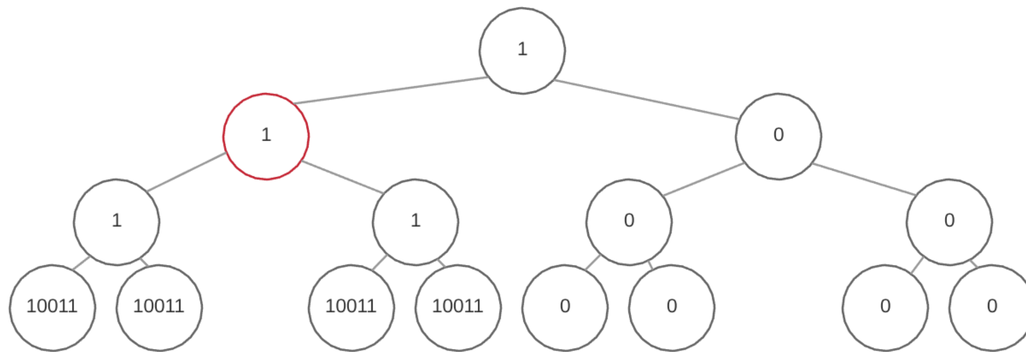

Figure 3.5. Allocation of node which is not a leaf (the red one) in the compact-level algorithm

seeing the situation of the bunch in its complex. In order to make it possible, in this version of the algorithm, when we do an allocation/free, we update the information also in the descendant and the ancestors of the node that are in the same bunch (an example of the allocation of a non-leaf node could be found in [Figure 3.5]); it is important to notice that also in this version, we do not propagate the information to the descendant of the node that are in other bunches. Finally, a single CAS tries to update the content of the bunch.

To understand if a non-leaf node is totally occupied it is sufficient to see if its leaf descendants in the same bunch are fully occupied or not; due to the fact that the granularity of a single CAS is "the bunch", it is sufficient, for the free (which was appropriately rewritten) only knowing if the leaf of the bunch is in a coalescing state.

3.3.1 Allocation

The allocation is possible by using a reviewed version of the algorithms *alloc()* and *check_parent()* and a new algorithm which is *occupy_descendants()*.

The first algorithm that I present is *alloc()*, Algorithm 8. This function takes in input a node, which is suspected to be free and it tries to allocate it. The function returns true if the allocation is successful; false otherwise.

In detail, the algorithm checks if the node is totally free (at line 4, remember that a node is allocable only if its value is 0) and returns false if it is not; then, if the node is free, it marks as occupied all the ancestors of the node that are in the same bunch (lines 7-9), and, in the lines 10-13 the algorithm marks the node n as occupied, moreover, if it has child in the same bunch, it marks them as occupied (lines 14-15). The control at line 14 is necessary to avoid segmentation fault on the last node if the last bunch is smaller than 4 levels. At line 17, the algorithm returns true if it is allocated a node in the root bunch; else the function *check_parent()*

is called to try to mark as allocated the ancestors of the node that are in parent's bunches; the result of *check_parent()* is returned.

If *check_parent()* returns false, before to return, the function, at line 20, restores the state as the operation was never called: to do this, the function has to know where the allocation failed; in this version of the algorithm, the upper_bound is a global variable which is set by the *check_parent()* function each time it is called; the *upper_bound* is the last node correctly allocated. Modify operation on the bunch (line 3-15) are done in a do-while loop to check concurrent modifications. In *alloc()* we only check if our node is occupied or it is free; we can do that because:

- if an ancestor is partially occupied, this means that a node on the other subtree is occupied; in this case we just set again the ancestor's bit and so we can not cause any problem;
- if an ancestor or a descendant in the same bunch is fully occupied, due to the fact that an allocation atomically updates all the bunch, we must see our node as occupied. By construction it can not happen that we see the node n as free and an ancestor, in the same bunch, is fully occupied.

Algorithm 8 Allocation

```
 1: procedure ALLOC(node* n) return bool
 2:     repeat
 3:         old_val = new_val = n → container → nodes
 4:         if !IS_ALLOCABLE(new_val, n → container_pos) then
 5:             return false
 6:         current = n; parent = parent(n)
 7:         root_of_bunch = ROOT(n)
 8:         while current != root_of_bunch do
 9:             new_val = LOCK_NOT_A_LEAF(new_val, parent→container_pos)
10:             current = parent; parent = parent(current)
11:         if IS_LEAF(n) then
12:             new_val = LOCK_A_LEAF(new_val, n→container_pos)
13:         else
14:             new_val=LOCK_NOT_A_LEAF(new_val, n→container_pos)
15:             if n has child in the same bunch then
16:                 new_val = occupy_descendants(n, new_val)
17:     until !CAS(&n→container→nodes, old_val, new_val)
18:     if n→container→root == TREE_ROOT then
19:         return true
20:     else if check_parent(ROOT(n)) then
21:         return true
22:     else
23:         free_node_(n)
24:         return false
```

The function *occupy_descendants()* presented in Algorithm 9 is an auxiliary function of *alloc()*. It takes as input the mask of nodes related to n's bunch and then marks as occupied all the descendants of the passed node n.

Algorithm 9 Occupy descendants

1: **procedure** OCCUPY_DESCENDANTS(node* n, unsigned long new_val) **return** unsigned long
2: **if** n is not a leaf of a bunch **then**
3: $new_val = LOCK_NOT_A_LEAF(new_val, left(n) \rightarrow container_pos)$
4: $new_val = LOCK_NOT_A_LEAF(new_val, right(n) \rightarrow container_pos)$
5: $new_val = occupy_descendants(left(n), new_val)$
6: $new_val = occupy_descendants(right(n), new_val)$
7: **else**
8: $new_val = LOCK_A_LEAF(new_val, left(n) \rightarrow container_pos)$
9: $new_val = LOCK_A_LEAF(new_val, right(n) \rightarrow container_pos)$
return new_val

The algorithm number 10 is *check_parent()*. This function does an important work. It takes in input a node n; this node, by construction of the alloc, which calls the function, is the root of a bunch.

The algorithm tries to set as occupied n's ancestors (that by construction are on another bunch, as n is a root node). In particular, for the leaf node of the parent bunch, it cleans the coalescing bit (if necessary) and sets the node as partially occupied, setting to one the bit related to n's branch; for the non-leaf block it sets the node as occupied. The function returns true if it is able to reach the global root; it returns false when it finds a block that it is fully occupied (the nodes that are partially occupied are not a problem, in fact it simply means that another descendant of the node is occupied). The possible cases that the function have to manage are:

1. it finds a node that has the bit of n's branch clear: the function simply sets the bunch's leaf as partially occupied on that branch and sets all the ancestor's bit;

2. it finds a node that is fully occupied: in this case, as said, the function has to return false. The fact that a node is fully occupied is explicit in the fact that the leaf node of the bunch has the fifth bit equal to 1;

3. it finds a node n that is partially occupied: this case is not a problem for the algorithm. This condition, recognizable by the fact that the blocks are occupied but the fifth bit of the leaf node is equal to 0; simply means that another node, descendant of n, is occupied. The algorithm sets again the "occupied" bit to be sure that a concurrent free does not clear those bits and it continues the recursion;

4. it finds a node that is in "coalescing": it is not a problem; the algorithm simply clean the coalescing bit and sets again the node as "occupied";

In the cases where the algorithm does not return false, it recursively calls itself to continue the allocation. *check_parent()* returns true when the global ROOT bunch gets successfully updated .

Algorithm 10 Check parent

```
 1: procedure CHECK_PARENT(node* n) return bool
 2:     upper_bound = n
 3:     parent = parent(n)
 4:     root_bunch = parent→container→root
 5:     repeat
 6:         new_val = old_val = parent→container→nodes
 7:         if IS_OCCUPIED(parent→container→nodes, parent→container_pos) then
 8:             return false
 9:         if n is the left child of parent then
10:             new_val = CLEAN_LEFT_COALESCE(new_val, parent→container_pos)
11:             new_val = OCCUPY_LEFT(new_val, parent→container_pos)
12:         else
13:             new_val = CLEAN_RIGHT_COALESCE(new_val, parent→container_pos)
14:             new_val = OCCUPY_RIGHT(new_val, parent→container_pos)
15:         update new_val setting all parent's ancestors in its bunch as occupied
16:     until !CAS(parent(n)→container→nodes, old_val, new_val)
17:     if root_bunch == TREE_ROOT then
18:         return true
19:     return check_parent(root_bunch)
```

3.3.2 Free

The algorithm 11 is the free function of our buddy system; actually it can be used both as a real free and as an auxiliary function to rollback the state after a failed allocation. We can divide the algorithm in steps:

1. in the first step, we set the coalescing bit (relative to the branch of the node that we are freeing) to 1. this is done by the function *mark()* presented in Algorithm 13; this is done only if we are in a bunch different from the root one. Notice that, this time, the coalescing bit is only set in the leaves of each bunch. *mark()* takes in input the root of a bunch, says n, and marks as coalescing the parent of n, which is, by construction, in another bunch;

2. in the lines (9-15) we set the ancestors of the node n being freed, that are on the same bunch of n, as free. This is done only if the brother of the current node is free; if not, the father of n must remain occupied;

3. in 17 we free the descendant of n that are in the same bunch (if there are);

4. finally, we free the node (lines 18-21) and we try to write changes using CAS;

5. at line 24 we clean the coalescing bit of the ancestor of n that are in other bunches; this operation is done by the auxiliary function *unmark()* presented in algorithm 14. We do not do this last step if n is in the root bunch or we already know that a brother of our ancestor is occupied.

Algorithm 11 Free

```
1: procedure FREE_NODE(node* n)
2:     if ROOT(n) ≠ upper_bound then
3:         mark(ROOT(n))
4:     repeat
5:         exit = false
6:         current = n
7:         parent = parent(n)
8:         old_val = new_val = current→container→nodes
9:         while current ≠ ROOT(n) do
10:            if the brother is in use then
11:                exit = true
12:                break
13:            new_val = UNLOCK_NOT_A_LEAF(new_val, parent→container_pos)
14:            current = parent
15:            parent = parent(current)
16:        if n is not a leaf and it has childs then
17:            new_val = free_descendants(n, new_val)
18:        if n is a leaf then
19:            new_val = UNLOCK_A_LEAF(new_val, n→container_pos)
20:        else
21:            new_val = UNLOCK_NOT_A_LEAF(new_val, n→container_pos)
22:    until !CAS(&n→container→nodes, old_val, new_val)
23:    if ROOT(n) ≠ upper_bound AND !exit then
24:        unmark_(ROOT(n))
```

Algorithm 12 Free descendants

```
1: procedure FREE_DESCENDANTS(node* n, unsigned long new_val) return unsigned long
2:     if n is not a leaf of a bunch then
3:         new_val = UNLOCK_NOT_A_LEAF(new_val, left(n)→container_pos)
4:         new_val = UNLOCK_NOT_A_LEAF(new_val, right(n)→container_pos)
5:         new_val = free_descendants(left(n), new_val)
6:         new_val = free_descendants(right(n), new_val)
7:     else
8:         new_val = UNLOCK_A_LEAF(new_val, left(n)→container_pos)
9:         new_val = UNLOCK_A_LEAF(new_val, right(n)→container_pos)
       return new_val
```

Algorithm 13 Mark

```
1: procedure MARK(node* n)
2:     parent = parent(n)
3:     repeat
4:         old_val = new_val = parent→container→nodes
5:         if n is the left child of parent then
6:             new_val = COALESCE_LEFT(new_val, parent→container_pos)
7:         else
8:             new_val = COALESCE_RIGHT(new_val, parent→container_pos)
9:     until !CAS(&parent→container→nodes, old_val, new_val)
10:    if ROOT(parent) ≠ upper_bound then
11:        mark(ROOT(parent))
```

In this version of the algorithm the function *unmark()* has become more complex, because it has to deal with the less information about the coalescing nodes.

In the *unmark()* function, the input node, n, is the root of a bunch and it is already free by construction of the *free()*, which calls the *unmark()*. This function will work on the parent's bunch.

1. In the [lines 7-8] and [lines 13-14] the function controls that the node that it is going to clean has the coalescing bit set, in fact, also in this version of the algorithm it could happen, as explained before, that the coalescing bit was freed by a concurrent allocation or a concurrent free. In this case the function will return;

2. If the coalesce bit is still set, the function cleans it [line 9 and 15] and it also cleans the occupied bit [lines 10 and 16]; it checks if the brother of n is occupied and, if it is, it jumps to the CAS;

3. In the lines 21-27 the function cleans all the ancestors that are present in that bunch. If it meets a node that have its brother occupied, it stops cycle and it jumps to the CAS;

4. a CAS tries to update the content;

5. if the function is not in the *upper_bound*'s bunch and it have not met a node with the brother occupied, the function recursively calls itself to continue the ascent.

In the end, I want to highlight the fact that, as seen before, it could happen that the function *mark()* marked as coalescing some nodes that the *unmark()* function might not be able to reach and they remain with the coalescing bit still set; this happens if a brother is in use. Also in this version of the algorithm this situation will not compromise the correctness because:

- the fact that the node is coalescing with the relative "partially occupied" bit does not means nothing different that partially occupied. Parents, which does not have coalescing bit, will simply remain with the "occupied" bit set; Allocations will not be possible neither for the leaves nodes, nor for the parent nodes;

- where a successive *free()* or *alloc()* on the child of the dirty nodes will traverse them, it will clean it. In the case of a *free()* function, the occupied bit of the ancestors will be also cleaned, in the case of an allocation, the bit of the ancestors will remain marked.

Algorithm 14 Unmark

```
1: procedure UNMARK_(node* n)
2:     repeat
3:         exit = false
4:         current = n
5:         parent = parent(current)
6:         old_val = new_val = parent→container→nodes
7:         if current is a left child of parent then
8:             if parent is not coalescing left then
9:                 return
10:            new_val = CLEAN_LEFT_COALESCE(new_val, parent→container_pos)
11:            new_val = CLEAN_OCCUPIED_LEFT(new_val, parent→container_pos)
12:            if IS_OCCUPIED_RIGHT(new_val, parent→container_pos) then
13:                continue
14:        else
15:            if parent is not coalescing right then
16:                return
17:            new_val = CLEAN_RIGHT_COALESCE(new_val, parent→container_pos)
18:            new_val = CLEAN_OCCUPIED_RIGHT(new_val, parent→container_pos)
19:            if IS_OCCUPIED_LEFT(new_val, parent→container_pos) then
20:                continue
21:        current = parent
22:        parent = parent(current)
23:        while current! = ROOT(current) do
24:            if the brother of current is occupied then
25:                exit = true
26:                break
27:            new_val = UNLOCK_NOT_A_LEAF(new_val, parent→container_pos)
28:            current = parent
29:            parent = parent(current)
30:    until !CAS(&parent(n)→container→nodes, old_val, new_val)
31:    if current!=upper_bound && !exit then
32:        unmark_(current)
```

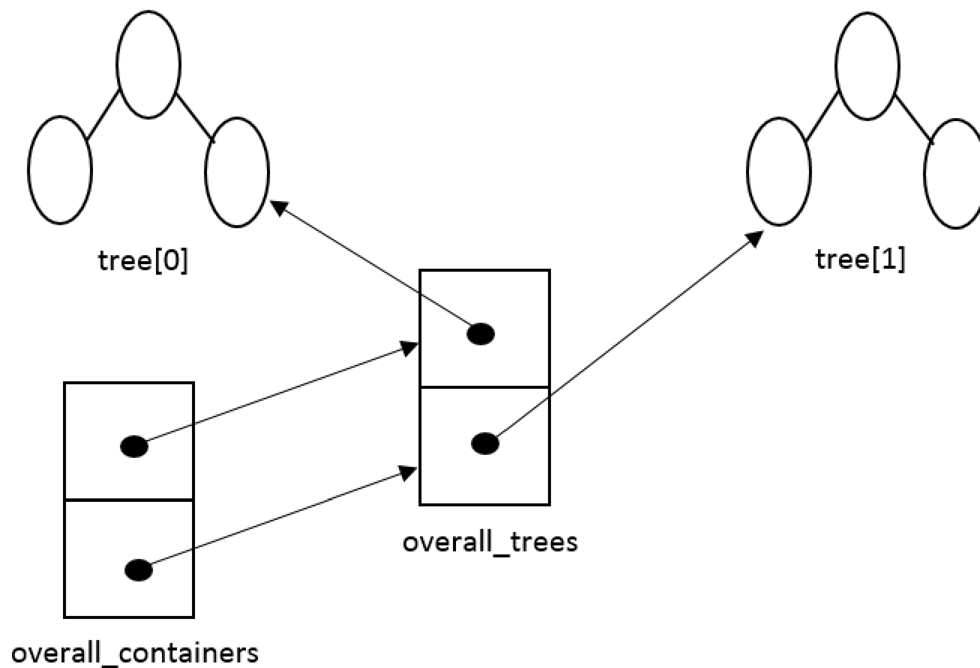

Figure 3.6. NUMA implementation

For what concern the ABA problem and the lock freedom property, considerations done in the first presentation of the algorithms naturally extends on this version.

3.4 NUMA implementation

NUMA implementation, as said before, it has be done by replicating data structures for each NUMA node. In particular, each NUMA node has its own tree and its own array of containers.

The actual implementation was done by defining three new global variables, which are array of dimensions equal to the number of NUMA nodes. These new variables are:

- *overall_trees*, an array of node pointers, in which each entry points to the root of a tree;
- *overall_containers*, an array of struct node_container pointers, in which each entry i points to the first instance of the i-th tree's container;
- the pointer that was used to point to the actual memory portion managed by the buddy systems was replaced by an array of pointers: each tree has its own memory block to manage.

A simple example composed by two NUMA nodes, with buddies formed by three nodes is shown in figure 3.6.

For what concern the system memory management it was not possible to use the the *numa_alloc_onnode()* function presented in [30]. This happens because memory allocated by this function is protected by Copy On Write (COW), this means that, if a child of the process which has started the buddy system, tries to modify a structure (in particular, container's values), the structure is replicated and the parent can not see any modification. This obviously lead to an inconsistent scenario in which the two processes actually work with two disjoint buddy systems referring to the same address space.

This problem was solved by using the classical *mmap()* function combined with the *MAP_SHARED* flag. The latter permits to parent which has called the *mmap()*, to share the memory with its child: modifications of the parent are seen by all the childes and vice-versa.

However, with *mmap()* you do not have the capability to control the node in which the memory will be allocated. For this reason, after the allocation, *numa_move_pages()* [30] is called. The latter permits to move pages to a specified NUMA node.

Chapter 4

Experimental data

In this chapter I present experimental data obtained running the algorithm. All the experiments have been carried out on top of a a x86-64 (which implies Total Store Order) machine composed by 24 cores running at 2.2GHz with 32GB of RAM divided in four NUMA nodes, running Linux (kernel version 2.6).

In particular, data are taken running the following algorithms:

- the one level, non-blocking algorithm presented in 3.1 ;
- the four levels, non-blocking algorithm presented in 3.3;
- a version of the one level algorithm which uses locks instead of non-blocking operations;
- ptmalloc3, which implementation is taken from the author's website [4].

These algorithms run the same tests, that consists in ten million of operations, each one that can be an allocation or a free with the same probability.

According with the buddy system definition, allocations are possible only with $2^i * MIN_BLOCK_SIZE$; for those experiments $i \in [0, 11]$.

Execution times are taken running tests with 1, 2, 4, 8, 12, 16, 20, 24 threads and in time-sharing with 48, 96, 128, 256 and 512 threads.

These tests have been conducted in three different scenarios:

- allocations possible from 8 B to 16 KB with the buddy systems managing 1 GB of memory (this means that the levels of the tree are twenty-eight);
- allocations possible from 2 KB to 4 MB with the buddy systems managing 1 GB of memory (this means that the levels of the tree are twenty);
- allocations possible from 128 B to 256 KB with a small buddy system managing 4 MB of memory (sixteen levels for the buddy system).

Configuration	Allocation size	Levels of the tree
1	[8B - 16KB]	28
2	[2KB - 4MB]	20
3	[128B - 256KB]	16

Table 4.1. Summary of the configurations used for test

A first, general consideration is that blocking algorithm suffers a lot of starvation: in all the runs, there are threads that end much sooner than others and this not happen for the non-blocking counterparts.

Another preliminary consideration is that for the blocking algorithm is possible to observe a greater variance in the running time between a run and another, identical, one. This could be problematic in real time systems and similar.

In the rest of the chapter I present in detail the obtained results.

4.1 Twenty-eight levels

In this test, allocations are possible from 8Bytes to 16KiloBytes and the buddy system is set up to handle 1 GB of memory. Results are depicted in figure 4.1

This scenario is appreciated by the PTMalloc because, as aforementioned in the introduction, it pre-reserves of memory for the blocks between 8 bytes and 64 bytes. In this case, it can use pre-reserved blocks for more than the forty percent of the cases.

This consideration, as expected, place the PTMalloc algorithm in a convenient situation.

Single thread execution In the single thread case the PTMalloc have great performances. This happen because it is very easy for the algorithm to manage a list of free blocks and, in the cases in which the memory ends, the request of new memory can be done without contention.

The second better algorithm is the blocking one, which is faster than the non-blocking counterparts. This result is expected and it will be common in all the testing cases. This happens because, as widely said in the introduction, atomic instructions have a non-negligible performances impact and, this algorithm uses them only for the pthread spinlock management.

Finally, we observe that the four levels algorithm is beaten from the one level. In this case, we can observe that the reduced number of CAS are not sufficient to mask penalties obtained by the complication of the algorithm; in particular, the

management of the containers could produce a negative cache impact.

The singled thread and the two core executions are the only in which I write about the blocking algorithm; increasing the threads number it pays the penalty of the blocking synchronization and its performances decrease in a way such that it is not comparable with the other competitors.

Multi thread execution Going parallel, we observe that the PTMalloc stabilizes its time on a threshold that could be obtained from the fast allocation served by the pre-allocated blocks, the request of new blocks to pre-allocate and the allocation that can not be served by the cache and that have to be executed in lock. The fact that a lot of allocation could be managed by the thread cache reduce the contention in a way such that it does not impact the performance.

The four levels algorithm is the one that improves more, obtaining better performances in respect to the one level algorithm in two cases (4 cores and from 20 to 24 core). Compare and swap becomes very costly when they are executed on data that are accessed by more threads at the same time: the reduced number of CAS for the four levels algorithm, actually balance the cost related to the higher algorithm complexity. However, the one level algorithm maintains very good performances due to the reduced number of retries respect to the four levels case.

4.2 Twenty levels

In the twenty levels case (whose results can be seen in figure 4.2), in which we have possible allocation from 2KB to 4MB, we observe that PTMalloc offers very poor performances. This happens because it can never use its cached allocator and, moreover, for request greater than 128KB it delegates the actual operating system to manage the allocation; in the other cases a locked queue is used.

In the first half of the chart we can see that the four levels algorithm is slightly better than the one level algorithm, the two curves intercept at 12 cores and, after that, the one level algorithm becomes clearly better. This means that the advantages obtained by the reduced number of CAS is not sufficient to contrast the higher complexity of the algorithm and, especially, the more conflicts obtained by collapsing levels in the same container. This effect was less important in the twenty-eight level case due to the lower density of the allocations, which are better distributed in a bigger tree.

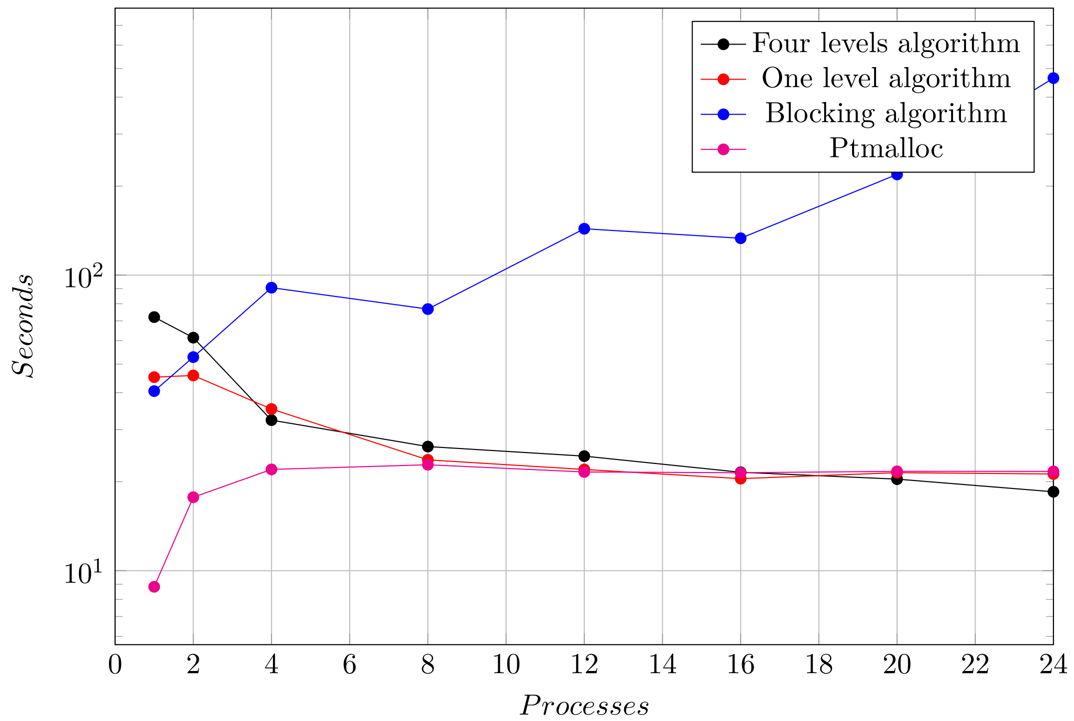

Figure 4.1. Execution time with 28 levels.

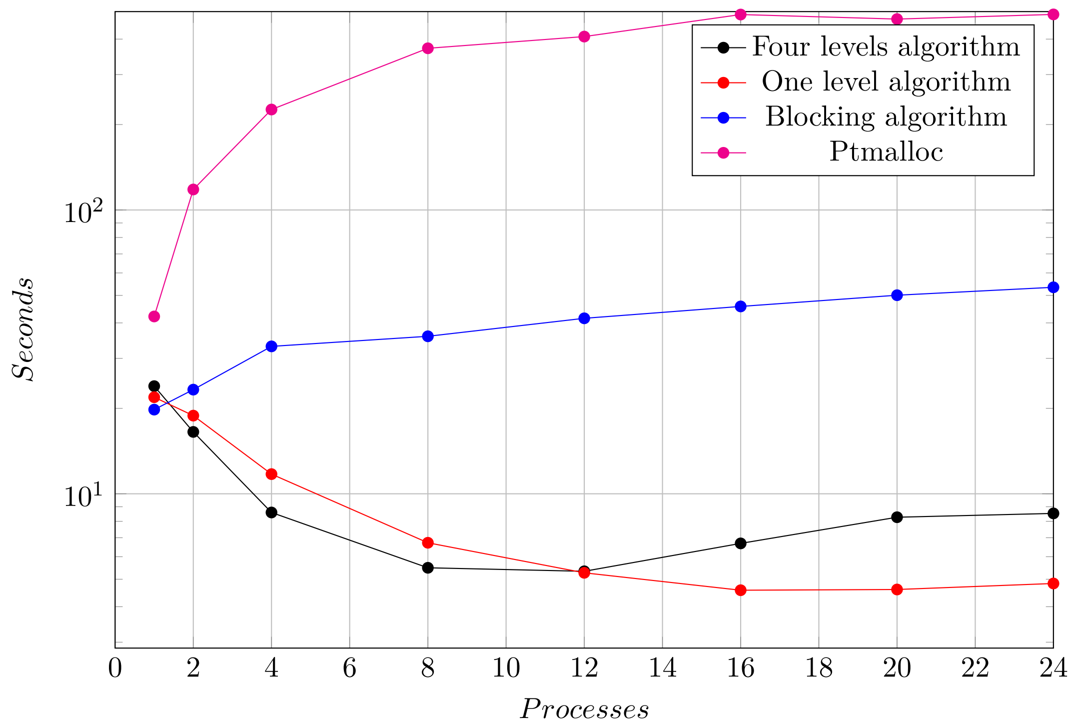

Figure 4.2. Execution time with 20 levels.

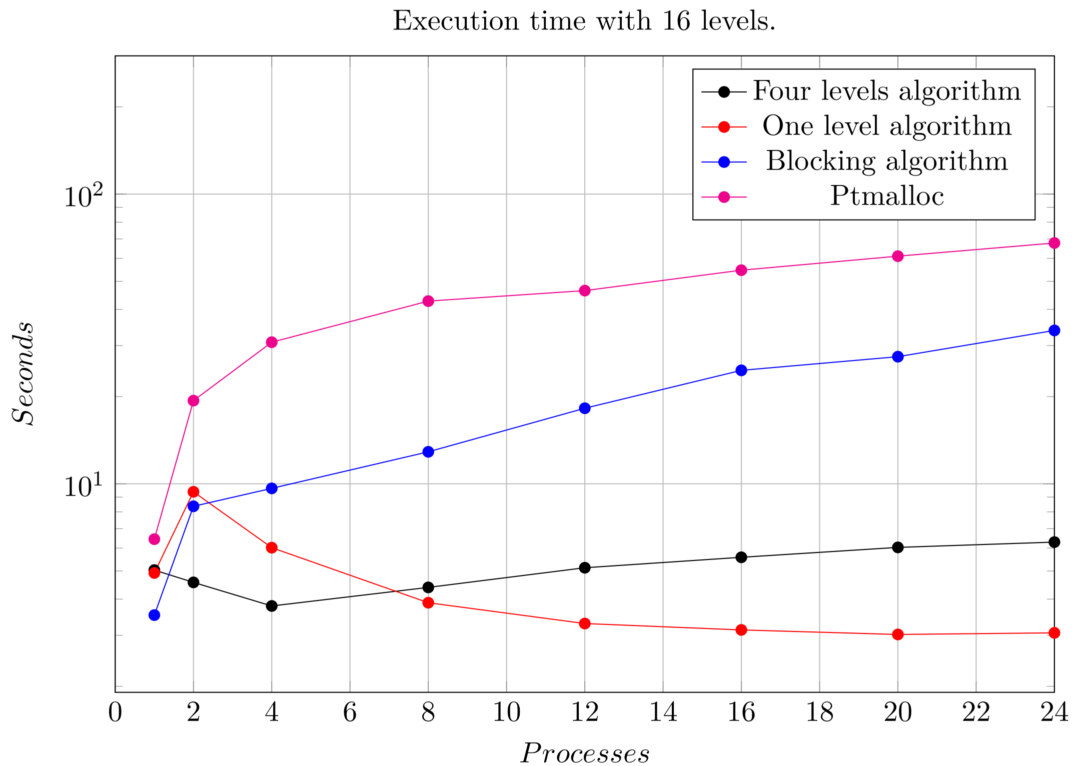

Figure 4.3. Execution time with 16 levels.

4.3 Sixteen levels

In the last case, depicted in figure 4.3, we have allocation possible from 128B to 256KB, PTMalloc is in a situation in which it uses all its options, in particular: it uses a sixth of the times the cached allocator or the operating system and two third of the time its locked queue. We have that its execution times are in general much better than before but not sufficiently to fill up the gap with the other solutions.

For what concern the two non-blocking algorithms we can see that the reduced dimension of the tree, as intuitive, causes small speed-up and, in certain situations we have higher execution times respect the serial case. In particular, this happens for the contention both in the upper levels of the tree, which are achieved with too few steps, and in the lower levels, in which the reduced number of nodes implies a great density of the operations. This contention is determinant with few cores for the one level algorithm and with more cores for the four levels algorithm. However, in general, in an environment which is not time shared, the trend seen in the first two cases can be confirmed, the one-level algorithm can, in general, have smaller execution time with smaller tree because of the reduced contention.

4.4 Time sharing

In this section I present the situation that we have in a time-sharing environment, i.e., the number of processes is greater than the number of cores, which, in this case, are twenty-four.

In this chapter I do not consider the blocking algorithm because its execution times are too higher for a useful comparative; also PTMalloc offers poor performances in the scenarios with allocations between $[2KB, 4MB]$ and between $[128B-262KB]$. PTMalloc works good in the situation in which its pre-reserving is relevant (allocation between $[8B-16K]$) and, for this reason, it is included in the charts only in that case.

In figures 4.4, 4.5, 4.6 the experimental data are plotted.

In the case, in which, as mentioned above, the PTMalloc is in its optimal situation, with the malloc size that fits its pre-reserved blocks, we can see that its performances do not degrade in an appreciable manner. However, despite the fact that we are in a PTMalloc convenient scenario, the two non-blocking algorithms remain competitive. In particular, the four levels algorithm offers slightly lower performance, while the one level algorithm beats the PTMalloc.

For what concern the twenty and sixteen levels cases there is not a real winner. Differently from the non-time-shared environment, in this case the four levels algorithm is worse than the one level algorithm with deeper tree. This could be related to the cost of the failed allocation, which are more probable in a time-shared environment, where an operation could be blocked in the meanwhile due to a scheduler (unpredictable) choice.

The very important thing is that, in all the cases, performances remain acceptable. This result is typical with wait-free algorithm and it is desirable writing lock-free algorithms. This does not happen neither with the blocking algorithm, nor with the ptmalloc, when it can not use caches and it has to run the actual parallel algorithm.

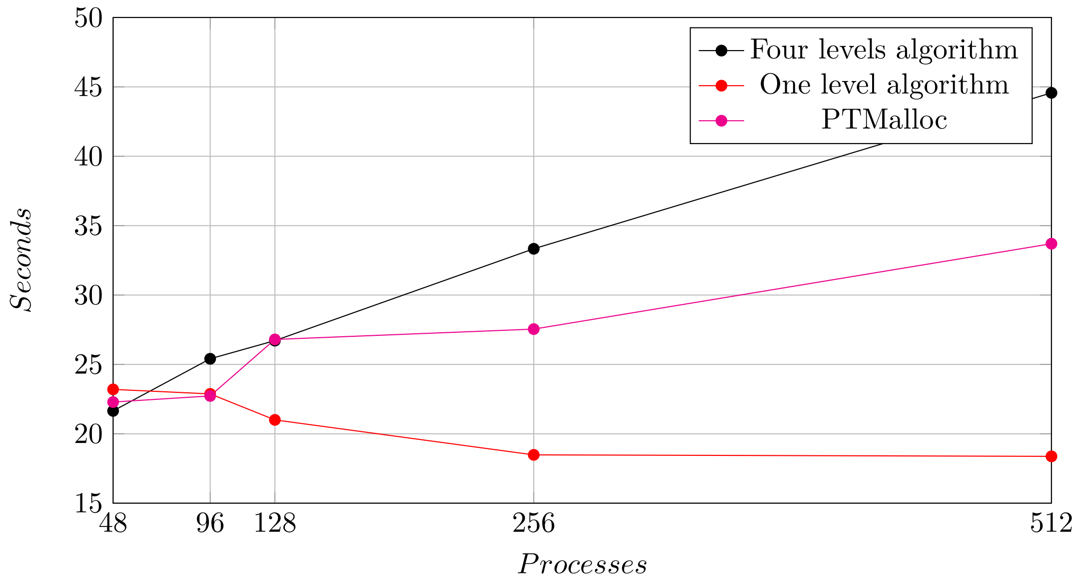

Figure 4.4. Time sharing results. Twenty-eight levels.

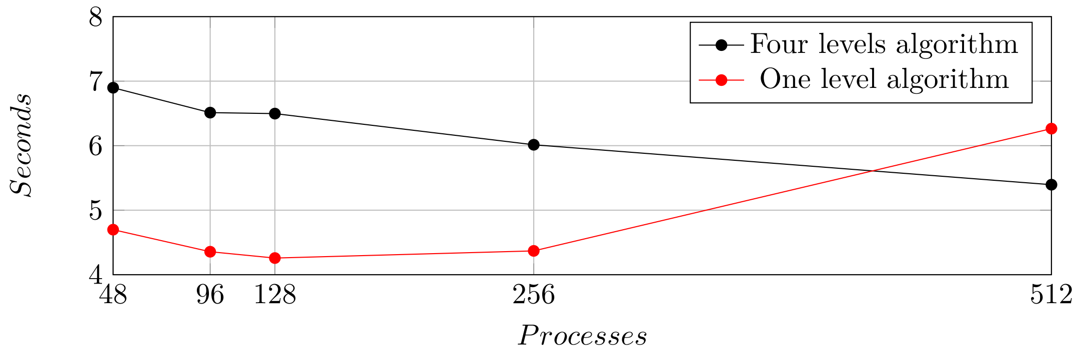

Figure 4.5. Time sharing results. Twenty levels.

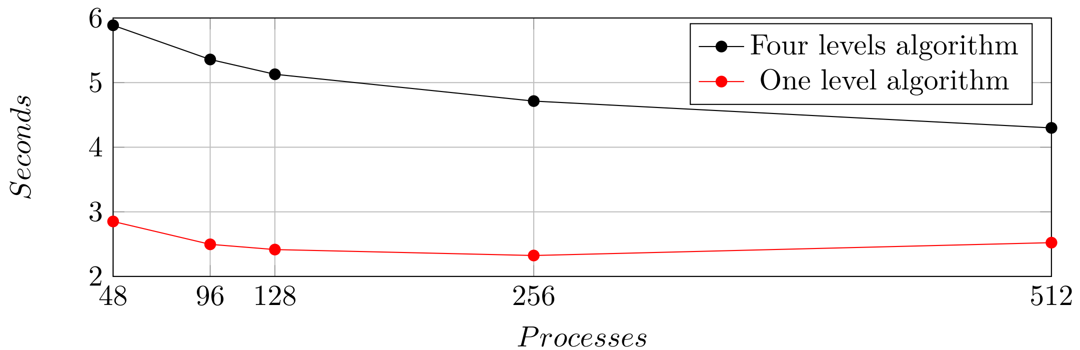

Figure 4.6. Time sharing results. Sixteen levels.

Chapter 5

Conclusions

In this work, I presented some advanced techniques related to parallel programming and I applied it to build a structure useful for the memory allocation.

Despite its limitations, that are related both to the fact that it is able to manage a fixed size of memory and to the bigger internal fragmentation in respect to the other algorithms; as aforementioned, buddy system is a methodology still useful in those situations in which it is necessary to have contiguous memory and performances represents a critical aspect.

In this work I presented the designing of a lock-free buddy system, explaining which are the steps which led to the final version of the algorithm.

With the experimental results, I shown that the proposed algorithm acts clearly better than standard implementations of the buddy systems, in which either a locked tree or free lists are used; in particular the experimental data shown that the algorithm maintain good performances in highly parallel, time-shared scenarios, in which the blocking counterparts becomes unusable. These results fully satisfy the initial objective to realize a fast, scalable memory allocator.

The experiments covered both the classical case of use in which the algorithm has to manage large space of memory for big allocations, and the scenario in which it has to satisfy small request of memory (i.e. memory allocation in message-passing applications).

An important consideration that follows from the experimental results is that it appears remarkable the impact of the cached allocation for the PTMalloc. This algorithm has behaved very badly in those circumstances in which it could not use that feature and, instead, it has very good performances in those situation in which the cache use was predominant.

Analyzing the effect of memory caching in a buddy system, which, by definition, has a limited amount of memory to manage, could be an interesting study to do as future work and, hopefully, to further increase performance.

Bibliography

[1] glibc wiki - malloc internals. https://sourceware.org/glibc/wiki/MallocInternals. Accessed: 2017-04-29.

[2] locklessinc - allocator benchmarks. https://locklessinc.com/benchmarks_allocator.shtml. Accessed: 2017-05-03.

[3] Sanjay ghemawat and paul menage. tcmalloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html. Accessed: 2017-04-29.

[4] Wolfram gloger's malloc homepage, ptmalloc. http://www.malloc.de/en/. Accessed: 2017-10-14.

[5] How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers, Computers, IEEE Transactions on, IEEE Trans. Comput*, (9):690, 1979.

[6] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[7] Juan Alemany and Edward W Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*, pages 125–134. ACM, 1992.

[8] Arvind Arvind and Jan-Willem Maessen. Memory model= instruction reordering+ store atomicity. *ACM SIGARCH Computer Architecture News*, 34(2):29–40, 2006.

[9] Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270. ACM, 1993.

[10] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGOPS Oper. Syst. Rev.*, 34(5):117–128, November 2000.

[11] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. Composing high-performance memory allocators. In *ACM SIGPLAN Notices*, volume 36, pages 114–124. ACM, 2001.

[12] Brian N Bershad. Practical considerations for non-blocking concurrent objects. In *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, pages 264–273. IEEE, 1993.

[13] Martin J Bligh, Matt Dobson, Darren Hart, and Gerrit Huizenga. Linux on numa systems. In *Proceedings of the Linux Symposium*, volume 1, pages 89–102, 2004.

[14] Colin Blundell, Milo MK Martin, and Thomas F Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 233–244. ACM, 2009.

[15] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.", 2005.

[16] D. Dice and A. Garthwaite. Mostly lock-free malloc. In *SIGPLAN Notices*, pages 269 – 280, Sun Microsystems Inc., Burlington, MA, USA, 2003.

[17] Brijesh Dongol, John Derrick, Lindsay Groves, and Graeme Smith. Defining correctness conditions for concurrent objects in multicore architectures. 2015.

[18] Faith Fich, Danny Hendler, and Nir Shavit. On the inherent weakness of conditional synchronization primitives. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 80–87. ACM, 2004.

[19] Timothy Harris. A pragmatic implementation of non-blocking linked-lists. *Distributed Computing*, pages 300–314, 2001.

[20] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215. ACM, 2004.

[21] M Herlihy and N Shavit. The art of multiprocessing programming. *Burlington, MA: Morgan Kaufmann Publs*, 2008.

[22] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[23] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[24] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[25] Maurice Herlihy and Nir Shavit. On the nature of progress. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, OPODIS'11, pages 313–328, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[27] Mark D. Hill. What is scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21, December 1990.

[28] Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. A wait-free multi-word atomic (1, n) register for large-scale data sharing on multi-core machines. *arXiv preprint arXiv:1707.07478*, 2017.

[29] Krainz Jakob and Schröder-preikschat Wolfgang. A wait-free dynamic storage allocator by adopting the helping queue pattern. 2010.

[30] Andi Kleen. A numa api for linux. *Novel Inc*, 2005.

[31] Kenneth C Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–624, 1965.

[32] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 223–234, New York, NY, USA, 2011. ACM.

[33] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. *SIGPLAN Not.*, 47(8):141–150, February 2012.

[34] Christoph Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40, 2013.

[35] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE transactions on computers*, (9):690–691, 1979.

[36] Doug Lea and Wolfram Gloger. A memory allocator, 1996.

[37] Nakul Manchanda and Karan Anand. Non-uniform memory access (numa). *New York University*, 2010.

[38] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. A lock-free o (1) event pool and its application to share-everything pdes platforms. In *Distributed Simulation and Real Time Applications (DS-RT), 2016 IEEE/ACM 20th International Symposium on*, pages 53–60. IEEE, 2016.

[39] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. A conflict-resilient lock-free calendar queue for scalable share-everything pdes platforms. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 15–26. ACM, 2017.

[40] Paul E McKenney. Memory ordering in modern microprocessors, part ii. *Linux Journal*, 2005(137):5, 2005.

[41] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

[42] M.M. Michael. Scalable lock-free dynamic memory allocation. *ACM SIGPLAN Notices*, 50(8):11–22, 2015.

[43] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *ACM SIGPLAN Notices*, volume 49, pages 317–328. ACM, 2014.

[44] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631, 1979.

[45] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[46] NM Pitman, F. Warren Burton, and EW Haddon. Buddy systems with selective splitting. *The Computer Journal*, 29(2):127–134, 1986.

[47] Scott Schneider, Christos D Antonopoulos, and Dimitrios S Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management*, pages 84–94. ACM, 2006.

[48] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[49] John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.

[50] David W Wall. *Limits of instruction-level parallelism*, volume 19. ACM, 1991.

[51] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. *Dynamic storage allocation: A survey and critical review*, pages 1–116. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[52] Emery D Berger Benjamin G Zorn and Kathryn S McKinley. Reconsidering custom memory allocation. 2002.

Ringraziamenti

Ora che questo lungo percorso è terminato, vorrei ringraziare tutte le persone con le quali ho condiviso i tanti momenti belli e che mi hanno aiutato nei momenti brutti.

Vorrei ringraziare innanzitutto la mia famiglia: i miei genitori, che mi hanno sempre incoraggiato e sostenuto, anche quando sembrava tutto impossibile e che hanno sopportato i miei momenti di "pazzia"; mia sorella e Domenico, per il sostegno e per avermi fatto recuperare il sonno, quando sembrava impossibile! :-); Luca, che quando vive con me penso mi stia odiando (p.s. comunque è colpa tua!).

Vorrei ringraziare gli amici di Isernia, quelli di una vita, i quali vedo purtroppo raramente, ma quando accade è sempre bello.

Grazie a tutti i "colleghi" universitari, a quelli con cui "ci vediamo il 2 Gennaio in biblioteca", con i quali, forse mettendoci ansia a vicenda, ce l'abbiamo fatta! In particolare grazie al polentone, Mirco e Simone con i quali ho passato giornate indimenticabili. (Ti voglio bene polentone <3).

Grazie per la sopportazione a tutti i vari coinquilini che si sono succeduti negli anni.

Ringrazio Simona, che è arrivata solo nell'ultimo anno e mi è stata tanto vicina, sopportandomi e incoraggiandomi tutti i giorni. Senza il suo aiuto in quest'ultimo periodo probabilmente sarei impazzito.

Infine vorrei ringraziare i professori Bruno Ciciani e Francesco Quaglia, per la passione che sono riusciti a trasmettermi per questo ambito dell'informatica e Mauro Ianni, Romolo Marotta e Alessandro Pellegrini (con cui ho instaurato un rapporto fantastico) per l'aiuto che mi hanno dato nella realizzazione delle tesi.