The parallelism shift and C++'s memory model

Johan Torp
johan.torp @ gmail.com

September 26, 2008

Abstract

The first part of the thesis is an overview of the paradigmatic shift to parallelism that is currently taking place. It explains why processors need to become parallel, how they might function and which types of parallelism there are. Given that information, it explains why threads and locks is not a suitable programming model and how threading is being improved and used to extract parallel performance. It also covers the problems that await new parallel programming models and how they might work. The final chapter surveys the landscape of existing parallel software and hardware projects and relates them to the overview. The overview is intended for programmers and architects of desktop and embedded systems.

The second part explains how to use C++'s upcoming memory model and atomic API. It also relates the memory model to classical definitions of distributed computing in an attempt to bridge the gap in terminology between the research literature and C++. An implementation of hazard pointers and a lock-free stack and queue are given as example C++0x code. This part is aimed at expert C++ developers and the research community.

Contents

Part I

A bird's eye view of desktop parallelism

Chapter 1

The shift

The hardware industry is telling us that we have to move to parallelism, even for ordinary desktop computers. This chapter explains why that is and how fast it is happening.

1.1 The need

Moore's law states that the number of transistors that can be fitted on a single integrated circuit will increase exponentially. So far it has held true but there is a constant debate on how long this will last. Former and present Intel representatives claim that this trend is likely to continue for 10-20 more years [10] [12]. Assuming this is true, the problem is what to do with all these transistors. Traditional single-core processors have encountered some problems, which makes it difficult to get more performance out of additional transistors. These problems are generally referred to as 'three walls'.

1.1.1 The memory wall

Processor clock rates have been increasing faster than memory clock rates has [33] and this trend is expected to continue [44]. This discrepancy was just getting worse and worse, meaning you got less and less performance out a processor clock cycle. The trend has been to add increasingly larger and faster cache memories. This helps alleviate the problem but does not solve it. Caches are expensive, take precious die space from the processor and introduce the complexity of cache misses.

1.1.2 The instruction level parallelism wall

Instruction level parallelism (ILP) is a measurement of how many instructions that can be processed simultaneously. There are many ways to increase ILP even on single core systems. A typical instruction consists of many small steps which each require a clock cycle to complete. To complete more instructions per

clock cycle most modern processors feature some kind of pipeline where multiple instructions are executed simultaneously. Finding sequential instructions which can be parallelized this way is difficult. One instruction can write to a memory location and the next might depend on this write to be completed. Run-time branching due to if-statements, object oriented polymorphism and execution control mechanisms make it even trickier.

Many techniques have been applied to extract ILP performance. Hardware branch prediction, hardware speculative execution (see §5.5.2), instruction reordering (a.k.a. out-of-order execution) and just-in-time compilation are some notable examples. In the end, ILP has a theoretical maximum of one instruction per execution unit and the limit is set by the processor clock frequency. In practice it is worse since the memory frequency cannot keep up. Superscalar processors can breach the one instruction per clock cycle wall by introducing multiple execution units per core. They still execute the same sequential program but have several parallel pipelines. However, these processors suffer even worse from data dependencies and increased need of branch prediction.

1.1.3 The power wall

Processors consume more and more power the faster they go. It is not a linear relationship. According to Intel it is "'quite typical to increase power by 73 percent simply to get a 13 percent improvement in performance"' [44] when overclocking. Underclocking on the other hand significantly increases instructions/power rate. By downclocking a processor by about 13% you get roughly half the power consumption [44]. The power wall stops us from increasing clock rates much further.

1.1.4 The performance of single core processors

Together these three walls make it very hard to extract more performance from single core processors. RapidMind, a company targeting parallel middleware, has compiled figure 1.1, which show some key parameters of single core processor:

The amount of ILP, measured in calculations per clock cycle, has flattened out over the last couple of years. So has the clock frequency. In other words, we have already experienced a big drop in performance increase for single core processors. On the upside, the number of transistors we can put on a single chip is still expected to grow exponentially for many years.

1.2 The idea of multicore

Until some new hardware technology replaces silicon, these three walls are here to stay. So a simple idea is to just take two processors, downclock them a little bit until they have about half the power consumption and get almost double the performance. When the processor clock rate falls, the memory wall problems

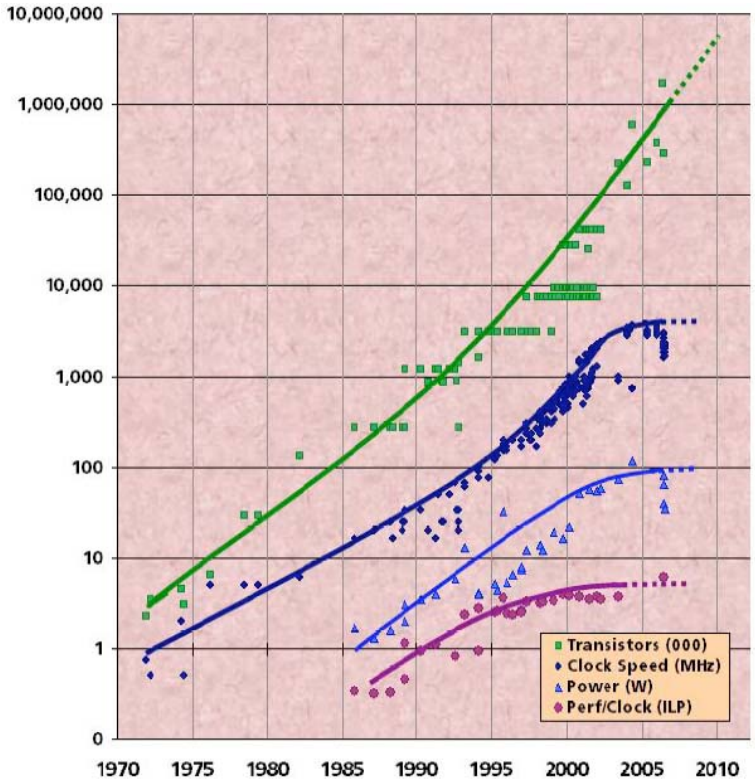

Figure 1.1: Single core processors.

become less noticeable. Therefore you get better performance per clock cycle and the clock rate decrease will not be very noticeable. This is the idea of *multicore*. Microsoft defines processors with up to 16 cores as multicore and processors with additional processors as *manycore*.

1.3 Manycore and heterogeneous processors

Multicore processors are already mainstream. The idea of manycore is to take loads of quite simple CPUs, put them on the same chip and create a massively parallel processor. Since many problems are somewhat inherently sequential and cannot be parallelized well it is likely that manycore processors will come with some larger, higher clocked cores and a lot of smaller ones. The manycore grid can be used for parallelizable problems, such as 3D-rendering. It is also likely that different cores will become specialized in certain domains such as integer or floating pointer arithmetic, logic and branching. Processors with several non-identical cores are called *heterogeneous*. Multicore and manycore processors are commonly referred to as Chip level MultiProcessors (CMPs) since multiple

distinct processors reside on the same chip.

1.4 Industry announcements

The hardware industry needs increased performance to motivate customers to buy a new machine. The software industry needs increased performance to enable new application areas and scale existing applications. As we will see later, parallel programming models force the traditionally rather independent industries to work together to deliver this performance.

Some of the major companies are sending out signals that we should prepare for a manycore future. In 2007 Microsoft released a document stating [38]:

> Microsoft and industry partners anticipate the advent of affordable general-purpose "manycore" systems in a few years.
>
> New processors will eventually come in heterogeneous configurations.

Intel's tera-scale research program states [45]:

> By scaling multicore architectures to 10s to 100s of cores and embracing a shift to parallel programming, we aim to enable applications and capabilities only dreamt of today.

AMD is targeting heterogeneous CMPs while Intel seems to be aiming at CMPs with as identical cores as possible [13] [52]. AMD's strategy seems natural since they have acquired ATI, a major producer of Graphical Processing Units (GPUs). GPUs are a form of highly parallel but domain specific processor, which are getting more generic.

Chapter 2

Parallelism and shared memory

A computer which runs many simultaneous programs can increase performance by running them on different cores. Unfortunately, we cannot take a single sequential program and just run it on multiple cores. To make it run faster, we somehow need to state which parts of the program can be run in parallel. And somebody needs to decide which parts shall be executed on which cores at what time. Cores will also need to communicate or *synchronize* with each other. On top of this, if cores *share memory*, careful thought must be given to how this memory is accessed, otherwise memory contention, bandwidth and latency issues might dramatically reduce performance. These factors fundamentally change the way to write code.

2.1 Concurrency and types of parallelism

Concurrency is modeling asynchronous actions so that they can be executed independently. For instance, you might start a separate thread to wait for a network response rather than have the main thread poll for a response. By modeling this with an additional thread, the program becomes cleaner and *conceptually* easier to understand. Concurrency is about abstraction and separating asynchronous concerns.

Parallelism is not about abstraction but about performance. It is about how programs are executed on parallel hardware to achieve high performance. It is also the focus of this thesis.

Parallelism is commonly divided into a few categories.

2.1.1 Task and data level parallelism

The most common distinction is between task level parallelism (TLP) and data level parallelism (DLP). TLP focuses on faster execution by dividing calcula-

tions onto multiple cores. DLP focuses on maximizing data throughput, for instance by dividing a program so that different parts access different memory ranges. TLP programs might execute different code on different cores while DLP programs might execute the exact same function on each element in very a large data structure. There is no clear separation between the two and a typical program exhibits both types of parallelism. Even so, the distinction is useful to express which performance bottleneck is being targeted - instruction execution time or memory latency and bandwidth.

2.1.2 Instruction and memory level parallelism

Task and data level parallelism should not be confused with instruction level parallelism (ILP) and memory level parallelism (MLP) which both work at a lower level. ILP typically refer to how a sequential program run on a single core can be split into micro-instructions. Multiple micro-instructions from subsequent instructions of the same program are then executed concurrently in a pipeline.

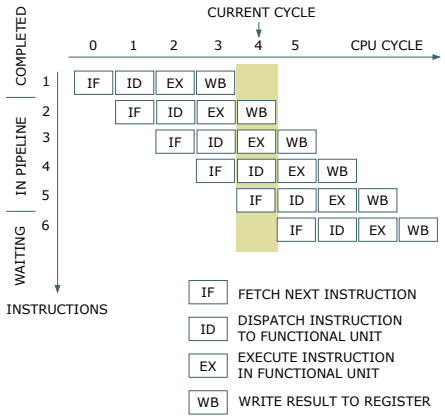

Figure 2.1: Simple pipeline with four micro-instructions per instruction

Because of inter-instruction data dependencies and run-time branching the amount of parallelism that can be extracted at this level is limited (see §1.1.2). MLP typically refer to how many pending memory operations, particularly the performance critical cache misses, a single core can have.

2.1.3 Bit level parallelism

At the lowest level is *bit level parallelism*. This refers to the word length which processors work at. It was a major source of speedup until 32-bit processors became mainstream and the increased precision and range of numbers were sufficient for most problems. Contemporary desktop processors usually operate on

64-bit data. The main drive behind the 32-bit to 64-bit shift was not computational power but rather the need to efficiently index more than 2^{32} bytes (i.e. 4gb) of memory.

2.2 The limitations of parallelism

The speedup factor is the time a particular problem takes to execute on a single core divided with the time it takes to execute on several cores. For instance, if a particular problem takes 1800 cycles to execute on a single core and 1000 on two cores, the speedup for two cores is 1.8.

Amdahl's law (equation 2.1) expresses the maximal theoretical speedup that can be achieved by executing a program concurrently on n cores. Synchronization costs are disregarded. It divides the amount of work into a parallelizable part and a sequential part. The proportion of work that is parallelizable is called p and the remaining sequential work is $1 - p$.

$$speedup(p, n) = \frac{1}{(1 - p) + p/n} \tag{2.1}$$

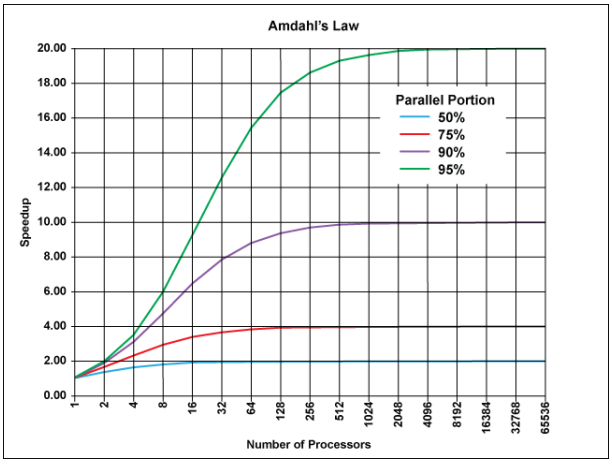

Figure 2.2: Amdahl's law. Courtesy of Wikipedia

Figure 2.2 shows the speed up for some values of p. With these gloomy results in mind, is manycore not a futile project? The speedup you will get from manycore will probably not be applicable to general purpose problems. There are however many useful problem areas which increase their parallelizable proportion the more you scale the problem. 3D-rendering at a varying resolution is a good example. The more you increase the resolution, the more pixels there are and more independent per-pixel operations which can be parallelized. To emphasize that many problems are scalable, Gustafsson's law (equation 2.2) was

created. It states that the speedup is a function of the number of processors, n, and the serial part of a program, α.

$$speedup(n) = n - \alpha \cdot (n - 1) \tag{2.2}$$

The serial part is constant with respect to the number of processors. It is assumed that there is always enough parallelism since the more cores you add, the more you scale the problem which in turns produces a larger parallelizable part. For these types of scalable problems, you can utilize and endless number of cores.

In general, DLP heavy applications often scale well by adding parallel computing power. TLP heavy applications, on the other hand, need to divide work into tasks. Because of runtime data dependencies, runtime branching and I/O operations (a.k.a. impure operations) many tasks cannot be executed in parallel (see §5.5.1 and §5.4.1). The granularity of task decomposition also puts a limit on scalability, i.e. how many cores you can add while still getting better performance. Fine granularity can be difficult or impossible to find, can be difficult to code and might introduce a lot of per-task overhead. You can often only approximate the execution time of a task so if a single task takes unexpectedly long time, the entire calculation can be delayed. Additionally at the start and end of many TLP algorithms all workers need to rendezvous before making some final calculations. This introduces a sequential part which cannot be parallelized.

2.3 Uniprocessor memory considerations

Having a high level idea of how memory works is important, even in high-level languages and single processor systems. By knowing some hardware basics, programmers can avoid the worst performance traps in performance critical parts of their code. Simply put, the hardware is not totally transparent. This gets a lot worse for multi/manycore systems but before considering them, let us revisit two performance critical hardware facts most programmers should be aware of.

2.3.1 Caching

Main memory operates at a slower clock rate than the processor and it usually takes a relatively long time to fetch data from it. To alleviate this problem, a processor often has a small amount of very fast memory which is located on the actual processor chip and can be accessed very fast. This memory is used to keep copies of the data the core is currently working with. As long as the processor only uses the data in the cache it can operate at full speed.

Caches are typically layered. A very small amount of super fast memory might reside on chip, another somewhat larger cache of rather fast memory is located near the processor while the main memory which is a lot larger and slower is located further away. The fastest memory is often called L1-cache

and subsequently larger and slower caches are called L2 and L3 if a third level exists. When main memory runs out, unused portions of it is often *swapped out* to some even larger and slower storage such as a hard drive.

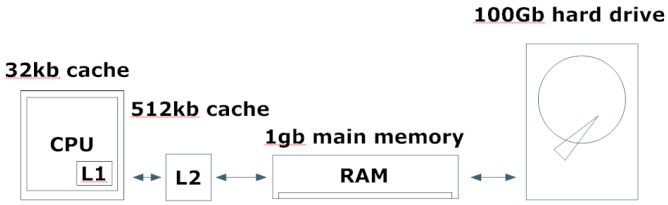

Figure 2.3: Example cache layering

When a processor accesses a memory location that is not already cached, a *cache miss* is said to occur. If an architecture has L1 and L2 caches you talk about primary and secondary cache misses. When a memory location which has been swapped down to external storage is accessed, a *page fault* occurs. Updating a cache after a cache miss can take 100s of CPU cycles. Accessing data from a hard drive often take 10ms which corresponds to 20,000,000 cycles on a 2Ghz processor. To get good performance it is crucial to avoid cache misses and page faults.

2.3.2 Locality of reference

During a short period of time, most programs access rather isolated chunks of memory. This phenomenon is known as *locality of reference*. There are two types of locality, temporal and spatial. Temporal locality states that if a program has accessed a memory location, it is likely to soon access the same memory location. Spatial locality states that if a program has accessed a particular memory location, it is likely to access nearby memory locations soon afterwards.

Locality of reference is very important for performance. You want the processor to work at its internal clock rate and not keep getting interrupted by cache misses and memory access latencies. A program which exhibits strong locality of reference can be greatly helped by caches and can even prefetch data which it believes is soon going to be accessed. Both caching and prefetching are techniques to battle the memory wall.

Locality of reference is not a new phenomenon. Mainstream programmers writing single-threaded programs have taken this fact into consideration for decades, for instance by accessing two-dimensional arrays in a contiguous way. Locality of reference is an example of how single processor hardware has not been totally transparent. As we will see soon, parallel processors are even less transparent. Greater and more complicated thought must be given to how memory is accessed in order to retain performance.

2.4 CMP shared memory considerations

Parallelism is not just about parallel computation. The computations need input data. Different processing nodes need to communicate and orchestrate a calculation too. Many parallel systems have been based on *message passing* to send input data, results and manage calculations between nodes. These kinds of systems are rather different than those which contemporary OSes, programming languages and programmers are adapted for.

The multicore desktop processors released so far have all had *shared memory* and it seems likely that this trend will continue. This is because today's desktop operating systems and the entire software stack is adapted for shared memory. The term shared memory means that at least some memory is accessible by all cores and that this memory is used for communication between cores. It can also refer to using memory for inter-process or even inter-thread communication on single-core systems, but the discussion in this chapter assumes more than one core. In this thesis the term refers to communication via shared memory; regardless if cores, processes or threads are communicating.

2.4.1 Parallel architectures

So how is memory and caches laid out in contemporary desktop multicore processors? One common type of architecture is called Symmetric Multi-Processing (SMP). In this scheme, each core is connected to the main memory by a shared bus.

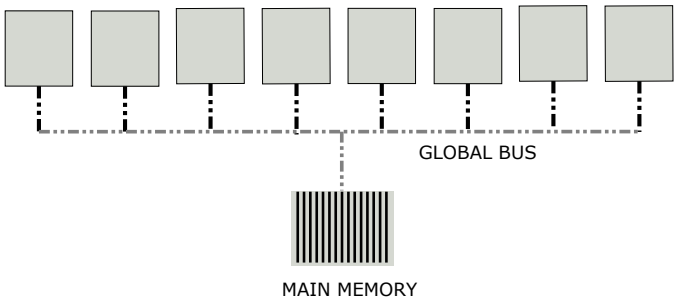

Figure 2.4: Example SMP architecture with 8 cores

Each core can have its own caches. Only one core at a time can use the bus which makes it a potential bottleneck. As the number of cores is increased, the situation gets worse. SMP architectures are common for contemporary dual

and quad core processors.

As more cores are added, more elaborate memory schemes pay off. Non Uniform Memory Access (NUMA) refers to any architecture where different parts of memory are faster to access by certain processors or cores. Each core can have its own main memory module. A couple of cores can also form a cluster which share a dedicated module through a local bus. Accessing memory belonging to other cores is done through some sort of global interconnect.

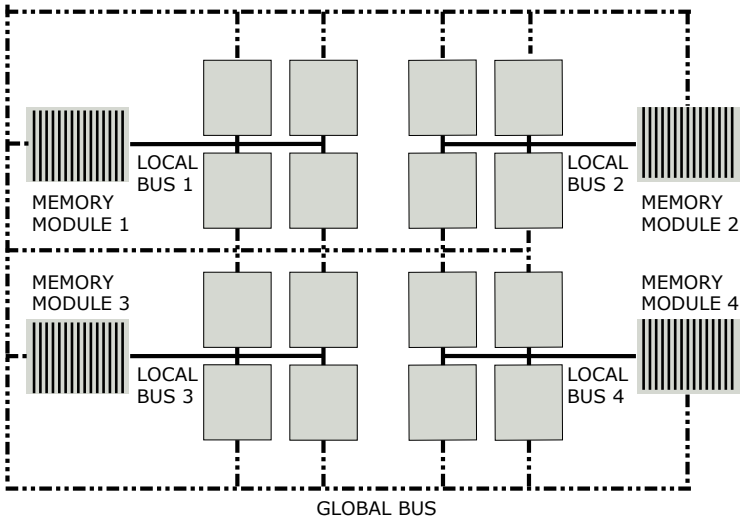

Figure 2.5: Example NUMA architecture with 16 cores and 4 memory banks

A bus-type interconnect can also become a performance bottleneck. More advanced interconnects can be used where access becomes more expensive the further away a memory module is. Section 4.3 of [Asanovic:EECS-2006-183] discusses what manycore interconnects might look like in more depth.

Cores could also have dedicated memory banks which they do not share by *reference* to other cores. Rather it can send *messages* to other cores which contain *copies* of memory chunks. At this point, we have moved away from pure shared memory architectures. Other promising architectures are focused on data flow and streaming data. This thesis focuses on shared memory architectures as they will probably remain the dominant desktop processors for many years to come.

2.4.2 The illusion and scaling of shared memory

Most shared memory models uphold the illusion that all memory is uniformly accessed. They typically do not model memory latencies, difference in laten-

cies for different memory locations, differences in latencies for different cores, bandwidth, interconnect contention or the size of the memory. Many operating systems even swap down memory to hard drives without telling you. Compilers and hardware go through great ordeals to provide this simplicity. Caches, hardware cache coherency protocols, out-of-order execution pipelines and memory interconnects consume very large parts of a processors die space (chip surface). Much of the time and money spent developing desktop hardware processors has gone to upholding the shared memory illusion. For single core processors, the problem has mainly been about easing and hiding away the memory wall. And it has worked fairly well.

With NUMA architectures we add some dimensions to the problem. Different cores can access different memory locations with different latency and bandwidth. Memory interconnects can also become a source of contention. As the number of cores increase and the interconnects get more complex, the performance of a simple shared memory programming model will get worse. At some point, *some part* of the software must begin considering the different latency, bandwidth and contention costs of accessing far away memory locations. Keeping memory and all cores on the same chip significantly reduces latencies, so CMP can probably take shared memory further than distributed setups.

2.4.3 Cache coherence

When you have multiple copies of the same data, you need to keep them in sync. This is rather straight-forward when you only have one core. When you have multiple cores with dedicated caches the problem gets more complicated. If one core increments the value of a memory location, this change needs to be propagated to all other cores' caches which contain copies of the value. This might take some time and meanwhile there is a risk that another core reads an outdated value and uses that to increment the value. We do not want to lose any increments but at the same time, we do not want to stop all cores that contain the value. Hardware cache coherence is an important puzzle piece in a *memory model* which is discussed in section 4.2.4 and second part of this thesis.

Maintaining cache coherence on NUMA architectures is inherently expensive. But since programming non-cache coherent NUMA architectures is very difficult, practically all NUMA processors contain some kind of hardware support to keep caches in sync. NUMA processors with hardware cache coherence are called cc-NUMA.

It seems unlikely that cache coherent shared memory solutions will scale to 1000s of cores [1]. On the other hand, shared memory is what contemporary OSes, programming languages and tools support. And what contemporary programmers are used to. It seems like a safe bet that industry will try to scale the shared memory model as far as possible.

2.4.4 Implications on shared memory programming

Multiple cores accessing the same memory and multiple caches require greater care, even in high level languages such as Java. This section considers a couple of common scenarios were programmers need to pay extra attention to the memory architecture of multicore systems. As always, these considerations need only be applied to the most performance critical parts of the code. Nevertheless, ignoring them can cause severe performance penalties.

Contention

As with any shared resource, care must be taken so that it does not become a bottleneck. When multiple cores all want to access some resource, we get *contention*. We have already seen that memory buses can be a source of contention. Likewise, if multiple cores all try to update the same memory location we can get contention problems. Avoiding various kinds of memory contention is a key to achieving core scalability.

Memory location contention

Contention can occur when multiple cores simultaneously access the same memory location and some of the accesses are writes. Keeping all caches in sync, making sure modifications are not lost and become visible in a timely manner can be quite expensive, especially on NUMA architectures. The programmer must think carefully so that no single memory location unintentionally becomes a bottleneck. The situation is similar to contention of traditional mutexes. Memory location contention can be avoided or minimized by selecting appropriate algorithms and concurrent containers.

Memory access contention

The memory bus of SMP and NUMA clusters is a common source of contention. For NUMA processors, additional memory interconnects are also a source of contention. Care must be taken to not access main memory when not needed and to rely on cached copies of data as much as possible.

Cache-line

Caches typically do not cache individual memory locations separately. Since the latency of fetching memory is much higher than the time it takes to transfer the content of a single memory location, it makes sense to copy several memory locations at a time. This way, locality of reference can help avoid future cache misses. Memory locations which are cached together are said to belong to the same *cache-line*. The size of cache-lines varies, but is often in the range of 8 to 512 bytes for contemporary processors.

Cache-line contention

A phenomenon that programmers need to be wary of is *false sharing*. It occurs when two different cores keep accessing and modifying two distinct memory locations which both happen to belong to the same cache-line. If core 1 updates any value in a cache-line which core 2 later accesses, then core 2 will get a cache miss. If both core 1 and core 2 keep modifying their respective values, both cores will suffer lots of cache misses and get greatly reduced performance even though they work with seemingly independent memory locations. False sharing can occur in other even less predictable situations. For instance, two independent node-based collections might find their nodes falsely shared by being located on the same cache-line.

2.4.5 Battling memory contention

As with any real engineering problem, there is no golden rule how to combat memory contention. Memory bandwidth or latency is often the factor which limits how far a problem can be scaled. Before reaching those hard limits, it is easy to accidentally add unnecessary contention problems to a shared memory program. This section includes a few guide lines how some of the pitfalls can be avoided. Most programmers can skip this section - the best approach to memory contention is to use concurrent libraries which handle it for you.

Separate and localize data

One way to prevent false sharing is to *pad* the most crucial memory locations with extra unused data to ensure that they wind up on separate cache-lines. In Java or C++, this can be done by creating a class with one member variable, which is actually used, and lots of dummy variables, which are never used. This raises the question how much padding is needed, which means some knowledge of the hardware is needed.

To maintain locality of reference it is still very important to localize data which is accessed frequently by the same thread. Herb Sutter provides three guide-lines on how to get a good memory layout [50]:

1. Keep data that are not used together apart in memory
2. Keep data that is frequently used together close together in memory
3. Keep hot (frequently accessed) and cold (infrequently accessed) data apart, even if the data belong to the same object.

Busy wait

It rarely makes sense to have busy-waiting loops in single core processor programs. For instance, if you are waiting on a mutual exclusion lock, it is often better to yield the current thread and let another thread execute which might

release the lock. When you have multiple cores, multiple threads are really running in parallel. If each core holds a mutex for a very small amount of time you can use a *spin lock* to avoid context switching. A spin lock is basically a busy-waiting loop on a shared flag. A core that wants mutual exclusion busy-waits until the flag becomes false and then enters the critical section by setting the flag to true. This needs to be done by some kind of atomic compare-and-swap or test-and-set instruction to ensure that two different threads or cores cannot set the flag to true simultaneously.

Avoiding bus contention at busy waiting loops

Spinning on a write operation leads to continuous main memory access. Consider a multicore processor where all cores tries to test-and-set the same mutex flag. As soon as one of them succeeds, the others are contending to access the memory. When the mutex flag is to be released and set to false this operation is hindered by contention. The expected write time is proportional to the number of cores which are contending for memory access. So, adding more cores will actually increase the total execution time.

Read operations, on the other hand, need not access the main memory as long as the cached copy is up to date. The unsuccessful cores should instead spin on a locally cached copy of the flag and let the hardware cache protocol notify them when there is a point in accessing main memory. This way, bus contention would only occur when the flag is being acquired.

Avoiding contention by exponential back-off

An atomic compare-and-swap operation which fail can be interpreted as a collision caused by some kind of contention has occurred. A common strategy when contention is detected is to wait a little longer before retrying. A good heuristic is to double your wait time each time a collision is detected. You also need to add some randomness to the wait time to ensure you are not keeping identical paces with other colliding parties. This strategy is known as *exponential back-off*.

Exponential back-off has been employed in many areas of asynchronous communication, for instance in Ethernet hardware and in TCP/IP software. It is a suitable strategy for spin locks too. It can also be used for concurrent containers such as stacks or queues. A client trying to busy-wait take an item from an empty collection can back-off and wait a while to let other clients put an item in.

Avoiding contention by elimination backoff

Concurrent stacks seem to inherently suffer from memory location contention - all parties which are pushing and popping must access the head node. As it turns out, we can extract more parallelism than the head memory location allows at the cost of approximating the FIFO property of a stack. The key thing to notice

is that a push and pops in some sense cancel each other out. Assuming that the number of pushes and pops are fairly equivalent, we can increase parallelism by letting pushing and popping parties cancel each other out. The stack needs to be implemented by some means that can detect a collision, such as a CAS or a timed lock acquisition. When a pusher or popper detects a collision they back-off to another collection. Here they look for a corresponding partner which they can exchange object with. If no waiting partner is found, they register themselves as waiting, wait for some time and then try to access the stack again. To not get the same contention at the waiting data structure, one of many rendezvous objects can be randomly selected and used for waiting.

This scheme is called *elimination back-off*. It is applicable to producer-consumer and rendezvous scenarios where it is ok to approximate ordering guarantees.

Chapter 3

Threads and locks as a programming model

The examples in the last section all had something in common. They all had some parameters which needed tuning. How much data should I pad a variable with to avoid cache-line contention? How do I make sure data is far apart? How long should I want before retrying in an exponential back off scheme? How many additional memory locations should be used in elimination backoff schemes? These kinds of questions abound in parallel programming and the usual answer is: "'It depends on the hardware"'. Unfortunately, choice of higher level abstractions such as algorithms and data structures also tends to depend on hardware.

Finding suitable programming models which are both easy to program and delivers good performance on most architectures is very difficult. Parallel programs cannot be rewritten and redistributed every time new hardware becomes available. New processors will continue to hit the market every year and hardware architectures will probably evolve rapidly too. There is a strong need to find viable programming models which:

- Are easy to program
- Are efficient in many problem domains
- Are portable
- Scale from multicore to manycore
- Are efficient on all common architectures
- Are abstract enough to allow future hardware innovations
- Whose components are composable with each other
- Whose components are composable with components from other programming models

Parallel programming has been researched for decades. Few people believe there is one single programming model which is suited for all types of problems. It seems likely that a number of programming models will be needed to extract parallelism from different problem types.

Explicit threads and locks is not a suitable programming model for most parallel problems. It is difficult to program, it is hard to scale from multicore to manycore, it is hard to make effective on various hardware architectures, it is not composable and it is very explicit. Threading is perhaps the most studied and best known parallel programming model. It is definitely the most criticized. As such, it is a good starting point to see where it errs and what we can do better.

3.1 Wording

First of all, it is important to clarify the wording. This section discusses the problems that contemporary OS threading implementations poses. It also assumes that a shared memory imperative programming language is used. The reader is assumed to know the meaning of the concepts *mutex, acquire, release, starvation, lock, locking, thread, process, wait, sleep, context switch, call stack, launch a thread and join a thread* in this context. The reader should be aware of *thread local storage*, which is similar to static or global memory but with one instance per thread. The reader should also understand that the operating system is in charge of *scheduling* processes and threads onto physical cores by either *time slicing* or running them in parallel on different cores.

The problems discussed in this chapter are not necessarily problems inherent in threads. The problems can be due to:

- Having explicit control over execution
- Achieving mutual exclusion by locks
- Shared memory model
- Imperative/sequenced programming model
- The OS managing and scheduling threads

It can also be due to combination of these factors. This chapter discusses problems with today's desktop OS implementations of threads rather than some abstraction notion of a thread. With this being said, most problems discussed really are inherent in abstract threads, locks and shared memory[1] and combinations of them. The next chapter explains how some of the problems can be fixed or lessened without replacing rewriting today's desktop software stacks from scratch.

[1] When the term shared memory is used in isolation, it typically refers to that separate cores or processes, not threads, communicate via shared memory. The related problems are the same though.

3.2 Difficult to program

Threads and locks programming has a few well known pitfalls which are difficult to avoid and even more difficult to detect and fix.

3.2.1 Deadlock

When several threads acquire multiple locks in different orders a *deadlock* can occur. The most basic example involves two threads, T1 and T2, which each acquire and release the same two mutexes, M1 and M2, but in different order. Assume T1 successfully acquires M1 and T2 M2. T1 then tries to acquire M2 while T2 tries to acquire M1. Neither thread will release their locks and both wind up waiting on each other indefinitely.

Thread 1 executes	Thread 2 executes
`acquire(mutex1);`	`acquire(mutex2);`
`acquire(mutex2);`	`acquire(mutex1);`
`// do something`	`// do something`
`release(mutex2);`	`release(mutex1);`
`release(mutex1);`	`release(mutex2);`

Table 3.1: Example program with risk of deadlock

More complex and unexpected situations can occur with more threads and locks becoming deadlocked in a circular fashion. Deadlocks can also occur when two threads are about to be joined if the thread which requests the join is holding a lock.

3.2.2 Not composable

The risk of deadlock can be avoided by having a global order which all locks must be acquired in. However, finding, maintaining and verifying that such an order is upheld is very difficult and introduces massive coupling. For the sake of modularity, you would like every component in a system to handle its own synchronization. But in many cases you need to perform indivisible (a.k.a. atomic) operations involving multiple objects which are protected by different mutexes. You then need to acquire multiple mutexes before performing the operation. This is impossible if every component hides away its mutexes. Lock hierarchies [49] can be used to partition the locks of an application according to existing layers. In practice, it only divides the problem into a couple of smaller parts which still suffer from the same inherent problem.

Ideally, you would like any component to be able to call any other component in the midst of a high level atomic operation. This is very dangerous in a lock-based program since you have no control of the order mutexes are being acquired.

3.2.3 Race conditions

A race condition occurs when two threads access the same shared memory location without proper synchronization. If one thread read a variable which another is simultaneously writing, non-sensical data might be read. Or the write might never be seen because it was only saved in some cache layer never visible to the reading thread. Another problem is knowing in which order writes to multiple variables become visible. Without having such guarantees, multithreaded programming is very difficult.

The most common approach to avoid a data race is to protect a shared variable with a lock - before accessing the variable you must acquire the lock. But which locks correspond to which variables are often left as comments. This makes it easy to forget to acquire a lock before accessing a variable.

3.2.4 Lost wakeup

Threads often need to sleep until a certain event or condition has occurred. A very common mechanism for waiting is called *condition variables*. One or more threads can wait on a condition variable and other threads can awaken, notify, one or all of the waiting threads.

A *lost wakeup* occurs when a thread goes to sleep and is never notified. One common scenario is to forget acquiring a lock before checking a condition. The waiting thread checks the condition and decides to wait and *then* acquires the lock. Before it manages to start waiting, the condition occurs and another thread notifies the condition variable. The waiting thread then waits forever.

Waiting with bug	Notifying thread	Correct waiting

`while (a_bool)`
`{`
`  lock l(a_mutex);`
`  a_cond.wait(l);`
`}`
	`lock l(a_mutex);`
	`a_bool = false;`
	`a_cond.notify();`
		`lock l(a_mutex);`
		`while (a_bool)`
		`{`
		`  a_cond.wait(l);`
		`}`

Table 3.2: Example of lost wakeup bug

Lost wakeup bugs can be hard to spot. The problem is that the condition variables programming model allows them to be expressed. Conditional critical sections, monitors, blocking reads, futures and transactional memory's retry statement are examples of conditional waiting constructs where lost wakeups cannot occur.

3.2.5 Indeterminism

Thread-related bugs are often notoriously difficult to detect and fix because threaded execution is inherently indeterministic. Data races occur only when two threads access the same variable at roughly the same time and even then

it might not cause any problems. Deadlocks might only occur in very special situations. Lots of thread-related bugs are rare and difficult to reproduce. This makes common testing techniques such as unit testing and regression testing hard to apply. The problem does not end here. A bug which practically never happens in a testing environment might happen all the time in a live environment, where it is executed on different hardware, with other user input or with a different network load.

3.2.6 Lack of memory isolation

Most contemporary operating systems have a concept of *processes*. A process has its own *address space* which is a chunk of memory not shared with other processes. This provides a level of *isolation* which tries to prevent that one faulty or malicious process from interfering with other healthy processes. An abstract threading model does not state anything about address spaces or isolation in itself - it is an orthogonal design decision. However, today's mainstream threading implementations do not provide any way of restrict which parts of the shared memory can be accessed by which threads. This lack of isolation can make it more difficult to write correct programs as any thread can affect the state of any other thread in unexpected ways.

Note that address restriction is implementable but would incur a performance penalty. Contemporary mainstream implementations do not provide any isolation safety net. Strong typing (not allowing type casts), not allowing global (a.k.a. static) data and purity (see §5.4.1 and §5.4.3) can help expose shared memory accesses and ease the problem.

3.3 Inefficient

Even though threads give you a lot of control, they are not very efficient in many aspects.

3.3.1 Cost per thread

Each thread has its own call stack, which consumes a lot of memory. All input and local variables for every level of method invocation must be stored. For OS thread implementations, each thread consumes kernel resources and the cost of management. The more lightweight a thread is, the more fine-grain task level parallelism can be extracted.

Contemporary implementations are fairly heavy weight. The default thread stack size on Windows is 1mb [39] and often 1mb or 8mb for Linux but can be configured. Some operating system, languages and libraries provide more lightweight alternatives. Naturally, the lightweightness comes at a price such as decreased isolation by using a shared stack or increased complexity by having to explicitly manage scheduling. Fibers, co-routines and thread pool libraries (see §4.6) are examples of more lightweight execution control mechanisms.

3.3.2 Cost of context switching

When a thread's time slice (a.k.a. quantum) runs out, it can be interrupted anywhere. While the thread itself does not notice, there is a lot happening behind the scenes. Registers need to be stored replaced with the next threads old values. The instruction pipeline needs to be emptied and refilled. The program counter needs to be replaced too. This all comes at a cost. But what is worse is that a context switch severely reduces locality of reference and thereby the instruction level parallelism.

If a thread is swapped out while holding the lock of an important shared resources, other threads risk being severely delayed. With classical locks, there is nothing which protects a thread from being swapped out while holding locks. This especially problematic in real time systems which supports thread priorities. *Priority inversion* occurs when a low priority thread blocks a high priority thread by holding a shared resource. If middle priority threads then starve the low priority thread, the high priority thread will never make any progress.

3.3.3 Does not address memory locality

The threading model does not convey any information on which memory locations it intends to access. Knowing how a program will access memory can help scheduling memory access in effective ways. Knowing which cores will communicate (access the same shared memory locations) with which others is also important for determining how threads should be laid out in space and time on the physical processor. Even though this information is often known by the programmer, the OS scheduler which schedules processes and threads does not have access to this information. At the same time, the programmer, who can choose the number of worker threads to start or use different memory access patterns, does not know what the hardware looks like.

Instead, the programmer must implicitly try to access memory in ways which might be effective. This can be done by starting a number of threads which is approximate to how many cores the programmer believes a typical user machine might run the program on and trying to maintain locality of reference on a per thread basis. Programmers can also implicitly assume that thread specific storage will be laid out on memory rapidly accessible by the core on which the thread is executed. Even if the implicit assumption leads to portable performance increases, the programmer's intentions often have to be expressed in comments which lead to maintainability issues. Contemporary threading models do not supply any features to explicitly declare how memory will be accessed.

3.4 Poor scalability

There are some obstacles which make it difficult to scale multithreaded code to manycore. In some sense, they are all due to the same problem: The threading model is too explicit while not knowing anything about the hardware it is being executed on.

3.4.1 Shared memory model

Any parallel shared memory programming model inherits the scalability problem of shared memory (see §2.4.2 and 2.4.3). Let us assume our code will be executed on some future NUMA processor. Today's threading implementations do not offer any ways to know or specify where some memory is located. You can probably assume that allocated memory is often mapped to physical memory located close to the core which executed the allocation. Likewise, you can probably assume that thread local storage is often laid out close to the core which will begin executing the particular thread. These assumptions might very well be performance critical. But they are implicit and can easily be misunderstood and broken by refactoring. Also, you cannot tell when fast localized memory runs out and your critical memory locations are laid out at distant locations.

3.4.2 Fix-grain task level parallelism

As explained in the section 3.3.1, starting a thread is a fairly heavy weight operation. Since the overhead is quite large, it does not make sense to use threads to represent a task. Assume I want to make 10 function calls which each take 1,000,000 cycles. If starting a thread takes 10,000,000 cycles, it is better to not start any extra threads at all, as that would slow execution down. The per-task overhead limits how fine-grain task level parallelism we can extract.

But it is not only the per-thread overhead which limits parallelism. You should also run as many threads as there are cores. If you start 1000 threads and only have 100 cores, the threads will probably executed in parallel by time slicing. This can severely reduce performance because each context switch has a cost and reduces locality of reference. If you only start 10 threads, only 10 of the 100 cores can work in parallel and you underutilize the hardware.

You could have a much lighter task abstraction than threads and divide your program into really fine-grain tasks. The tasks could then be executed in some kind of thread pool (see §4.6). But there is also a programming burden to explicitly divide work into a number of subtasks. And this burden is often greater, the finer granularity you require. Even worse, if individual tasks access the same shared memory you would still have to explicitly reason about synchronization and memory contention.

3.4.3 Fix-grain synchronization

Threading not only forces programmers (or some library) to explicitly state how many parallel threads of execution there should be. When those threads access shared memory the programmer also has to decide at what granularity he wants synchronization to occur.

One single mutex can be used to protect all shared data in an entire application. If a lot of shared memory is accessed, the mutex would become a great source of contention. In essence the program would become sequential. It would also introduce massive code coupling, just like any global data.

On the other extreme, we could use really fine-grain locking and have one mutex per memory location. This would minimize the time threads spent waiting on each other. On the other hand, because of the overhead of acquiring and releasing a mutex the program might spend most of its time acquiring and releasing locks. What is worse, it is often notoriously difficult to get fine-grain locking right and it also tends to couple your program (see §4.1).

3.5 Dormant bugs

Threads and locks related bugs manifest themselves when threads interleave in dangerous patterns. On single core processors, context switching is rather expensive and threads do not interleave much. On CMPs there are lots and lots of interleavings between threads. And probably new patterns of interleavings too. There will be more interleavings, the more cores you add. There are probably huge amounts of thread-related bugs out there, which have only manifested themselves very rarely, if ever. As we turn to multicore, software which has worked fine for years on single core processors might become unstable.

Chapter 4

Threading for performance

The contemporary threading model is not a desirable parallel programming model. But it is deployed and widespread. And, perhaps more importantly, well understood by millions of programmers. There is tremendous value in all the thread-based systems out there and we cannot rewrite them over night. What we can do is incrementally improve the threading model while introducing other programming models in parallel. This chapter explores how parallelism can be extracted from thread-based library solutions and how the threading model is being enhanced.

4.1 Fine-grained locking

To extract thread-based parallelism, threads need to run in parallel. It is crucial that threads do not spend large amount of time waiting on each other. Fine-grained locking tries to minimize the time threads spend waiting on each other by having lots of mutexes which each protect a small amount of data. The idea is to minimize unnecessary mutex contention on actions that could be run in parallel. For instance, a concurrent container class can keep one lock per item held. This allows many threads to access the container simultaneously, as long as they access different elements. Fine-grain locking is often notoriously difficult to get right. Much research has been put into finding and proving the correctness of fine-grain lock-based algorithms. The results from this research can and have been packaged into libraries. The main problem with that approach is composability.

4.1.1 Composability of lock-based containers

Consider containers. In general, there is no way to atomically move one element from one lock-based container to another. Both inserting and removing an item requires acquiring at least one mutex. First of all, to even have a chance of implementing an atomic move those mutexes must be taken together. This

means the locking must somehow be exposed. As soon as you expose locking you risk that users will violate locking orders and cause deadlocks. You can expose the locks to library internal locking schemes. This way you can implement atomic moves within containers of the same library but you cannot combine them with other libraries.

4.1.2 Composability of locking orders

The composability problem is not just about containers. In general, it is impossible to enforce two arbitrary locking orders. Having a global locking order including all mutexes in a program might seem like a good way around this problem. First of all, a global order usually introduces a high degree of coupling. Unrelated parts of a system gets a dependency to a single order and you still cannot compose with third party components. One could use the value of the mutexes' addresses as an order to side-step the code coupling problem. Still, the knowledge of the locking strategy will be spread over the entire system in an implicit and dangerous way.

Secondly, having lock orders severely limits the type of code you can write. You first need to decide which mutexes to acquire, then lock them in the correct order, then do something. This can have far reaching and awkward effects on program structure. Even worse, to know which mutexes you need, you might have to acquire a mutex.

```
bool a, b, c;
// Protects a, b and c respectively
mutex a_mutex, b_mutex, c_mutex;

// Here mutexes should be acquired before reading a, b and c

if (a) do_stuff(b); else do_stuff(c);
```

Assume some locking order states that b's and c's mutex should be acquired before a's. But to know which one of them to acquire, a_mutex must be acquired first. So there is no way to read a, b and c atomically without violating the lock order.

4.2 Atomic operations and reorderings

A shared memory model requires us to think about memory *consistency*. Unfortunately it is not enough to have cache coherent memory. We typically want a number of operations to happen *atomically* too. Imagine two threads which both read a value, increment the read value by one and then write it back. After both threads have executed we expect that the value to has increased by two. If thread 1 and 2 both read the same value before the other one performed its write, the value would only increase by one. If the read, increment and write could not be interleaved by the other thread's operations, the problem would be solved. A solution is to protect the shared value with a mutex. The mutex is acquired before reading the value released after the write.

4.2.1 Atomic instructions

Locks are not only dangerous but can be costly to use too. Acquiring, releasing, and waiting on a lock are all somewhat expensive. This limits how fine-grain parallelism can be extracted. A popular and more lightweight approach is *atomic instructions*. It is a single hardware instruction which promises mutual exclusion. The locking is there but it is performed at the hardware level which makes it really fast. In some sense, each memory location has its own lock. These locks are not exposed in any way so you cannot hold two of them at the same time. Therefore, atomic instructions can never deadlock[1]. On the other hand, you have to find another way to compose multiple instructions into an atomic operation.

Most hardware supply some *read-modify-write* (RMW) instructions. Increment is a typical one. It reads a value, increments it by one and writes it back in one single atomic hardware instruction. Two such instructions executed in parallel are guaranteed to increment a value by two. RMW instructions are very interesting since some of them can be used to construct higher level abstractions. For instance, you cannot implement a practical mutex[2] in terms of only read and write instructions [23]. Compare-and-swap (CAS) is a RMW instruction which is well suited for building higher level abstractions.

4.2.2 Reorderings

A compiler can perform substantial optimizations by moving around statements. This can be very dangerous in a multithreaded setting. Consider what happens if the compiler switches places of a lock acquisition and a later access to a shared variable. This reordering is safe from a single-threaded point of view but it is everything but safe when multiple threads are involved.

The hardware has also got good reasons to perform reorderings. It wants to increase the ILP. Memory accesses are expensive, especially when cache misses occur (see §1.1). To avoid being stalled too much, processors can queue up instructions until all their input data is available. Instructions are then executed one by one when they are ready. Later instructions are allowed to execute before earlier ones which are not ready yet. This is called *out-of-order execution* and is very common in contemporary processors. The hardware is smart enough to not reorder instructions so that program behavior is changed. For instance, a write and a read to the same memory location will not be reordered. But in a multithreaded setting, the compiler cannot tell what is safe and what is not by just looking at the instructions of one thread. It needs more information.

[1] Actually, in the presence of signal handlers and similar interrupts it is not enough to not encapsulate mutex locking so that no other mutex is acquired before calling unlock.

[2] A general mutex which handles an arbitrary number of threads in constant space. This cannot be implemented even in an ordered execution. For more information, see consensus numbers [23]

4.2.3 Memory barriers

How does the compiler and hardware know what reorderings are not allowed? You can tell the compiler that no reorderings are allowed by putting a memory barrier in the code. The compiler makes sure none of its optimizations violate this constraint. The compiler also emits a special instruction to the processor which it can use to flush its pending instructions.

Locks have these kind of barriers in their implementations. So locks actually do two things for us, it allows us to build arbitrarily complex atomic methods which appear to happen one at a time and it also prevents harmful reorderings. Special barriers can allow particular reorderings, such as moving a statement below an acquire lock statement but never above it. Memory barriers are also called memory fences or just membars.

4.2.4 Memory model

Heavy use of explicit memory barriers kills a lot useful optimization and ILP. *Memory models* allow a more implicit approach to reordering constraints. They describe how different threads interact by writing and reading shared memory, that is how a multithreaded program should behave. This can give the compiler and hardware a lot of more reordering freedom. Memory models can be a rather mind-bending topic and can be avoided by most programmers by relying on higher level abstractions such as locks or concurrent libraries. Memory models in general and C++ upcoming memory model in particular is the topic of the second part of this thesis.

4.3 Lock-free programming

Lock-free programming builds on really fast atomic instructions. It can be used to maximize the throughput of a central component or to increase its responsiveness. Lock-free programming has also got a nice progress condition which helps avoid some thread-related problems. But lock-free components are very difficult to write and can be even more difficult to extend and compose into higher level atomic operations. Many lock-free algorithms are covered by patent claims.

4.3.1 Progress conditions

Non-blocking synchronization commonly refers to one of three progress conditions, *wait-free*, lock-free and *obstruction-free*. A wait-free algorithm guarantees that any operation finishes in a finite number of steps, regardless of what other threads are doing. A lock-free algorithm guarantee that at least some thread is constantly making progress, regardless of what other threads are doing. An obstruction-free algorithm guarantees that any thread finishes in a finite number of steps if all other threads are idle. Looking at the definitions, we see that wait-free implies lock-free and obstruction-free and lock-free implies obstruction-free.

Obstruction-free algorithms can never deadlock but can *livelock*[3]. Lock-free algorithms guarantee that livelocks can not occur while wait-free go even further and guarantee that no thread is ever starved. Practical wait-free algorithms are difficult to find. Therefore, more attention has been turned to lock-free algorithms. They do not provide as strong theoretical guarantees but may perform very well in practice.

4.3.2 Compare-And-Swap

Non-blocking synchronization usually relies on the presence of certain atomic instructions (see §4.2) such as CAS. This hardware instruction is often available as an atomic operation in a high level language. A CAS atomically compares the value of a memory location to a given *comparand* value and, if they are the same, writes a new value. The instruction usually takes three value, a reference to the memory location it operates on, the comparand and the possible new value.

4.3.3 Stack example

A common example of lock-free programming is a simple parallel stack which is implemented as a singly linked list.

```
1  // Example CAS interface. Returns true if CAS was successful.
2  // Does not return the old value of target_reference.
3  template<class U>
4  bool compare_and_swap(U& target_reference,
5                        U comparand_value,
6                        U new_value);
7
8  template<class T>
9  class stack {
10
11   class node {
12   public:
13     T item_;
14     node next_;
15   };
16
17   node head_;
18
19 public:
20
21   void push(T item)
22   {
23     node* new_node = new node();
24     new_node->item_ = item;
25     do {
26       new_node->next_ = head_->next_;
27     } while (!compare_and_swap(head->next_,
```

[3]Livelock occurs when two or more threads keep interfering with eachother so that none of them make progress. Consider two people that fail to pass eachother in a narrow corridor because they both keep stepping aside in a completely synchonous fashion.

```
28                                      new_node->next_,
29                                      new_node));
30    }
31
32    T pop()
33    {
34      node* popped_node;
35      do {
36        popped_node = head_->next_;
37        if (popped_node == null) return T();
38
39      } while (!compare_and_swap(head_->next_,
40                                      popped_node,
41                                      popped_node->next_));
42      T popped_item = popped_node->item_;
43      delete popped_node;
44      return popped_item;
45    }
46  }
```

Listing 4.1: Naive lock-free stack implementation

A stack class has a dummy head node whose item is never valid or used. The head node's next field is always pointing to the top item in the stack. If it is null, the stack is empty. When pushing an item on the stack, a new node is created to hold the item. The next field is then repointed to head and finally we try to replace heads next pointer to point to our new node. This is the interesting part.

We atomically want to do two things: Set `node->next_` to the old top item `head_->next_` and replace `head_->next_` with our new node. In other words we want to perform two instructions on two separate memory locations (`node->next_` and `head_->next_`) atomically. If we did not take care to do this atomically, two threads might read the same top item, point their next node to that item and then both write to head's next field. This would cause one of the inserted nodes to be lost, as explained in figure 4.1.

But atomic instructions only operate on a single memory location so there is no way we can execute these two operations atomically at the hardware level. What we can do is at the core of lock-free programming. We make a conditional write to `head_->next_`, one that does not take effect if `head_->next_` has already been repointed. If someone else beat us to the update, we simply repoint our next node to that new node and retry the operation.

Starvation is possible; One thread can keep spinning if other threads are constantly updating the head node. But the important point to note here is that some thread is always making progress. If one CAS fails, another one CAS is bound to have succeeded[4]. This means some thread has made progress.

The pop implementation is provided to give a complete understanding of the class. Careful readers will note that this implementation suffers from a dangling

[4] Actually, some systems allow *spurious failures* for CAS implementations even if the comparand and memory location match. These failures are rare but sometimes code must be adapted to consider this fact.

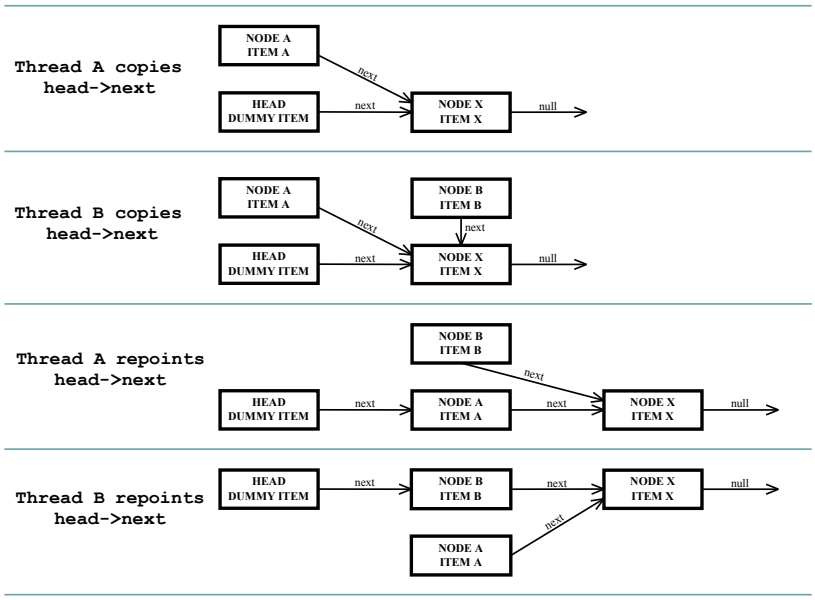

Figure 4.1: Example where non-atomic insertion causes node A to become lost

pointer problem. One thread might be dereferencing a particular node, while another thread deletes it. The problem of safely deleting shared pointers is called *memory reclamation* and is discussed in section 4.4.

4.3.4 ABA problem

One thing that complicates lock-free programming is the *ABA problem*. The problem is intrinsic in instructions like CAS where a value comparison is used to indicate that nothing has changed. Other threads can execute between the initial read and the verifying CAS. These threads can update the value multiple times and the set it back to the original value. So *something* might very well have happened, even though the value has not changed. The name comes from a variable with an original value A that is changed into a value B and then back to A again.

The ABA phenomenon is present in the stack example above, but if you think about it, it does not cause any problems. If the top node ($head\_- > next$) has been deleted and replaced by a new node with the same address, the node being inserted will still point to the new node.

Load-linked and store-conditional (LL/SC) are two other instructions which can be used instead of CAS to prevent the ABA problem. You first read a value by loading a link. When you later update the value via store-conditional, the

hardware has been watching the memory location for you and aborts the store if it has been modified since the initial load.

4.3.5 Lock-free memory allocation

Another important problem in the stack example is the use of **new** and **delete**. A default implementation of the new allocator is probably lock-based and would break the lock-free property. Care must be taken to use an allocator that is suited for multithreaded allocation (see §4.4.6). Example of lock-free allocation strategies can be found in [46], [16], [35] and [11].

4.3.6 Helping

Sometimes the composed operation you need to execute atomically cannot be expressed by a single CAS loop. An interesting technique to implement more advanced atomic operations, which consist of multiple atomic operations, is *helping*. Say you have an operation which requires two CAS operations. If the first CAS succeeds you move on to the second. But when the first CAS fails, you know that someone else is executing an atomic operation. If you have the appropriate information you can then try to help the unfinished thread by executing the second CAS for it. This ensures that the unfinished thread makes progress even if it has been swapped out or has crashed. After helping the unfinished thread you go back and retry your first CAS. See [36] for further information and an example.

4.3.7 Waiting strategies

In more advanced lock-free implementations, a contention manager can decide what you should do when you fail completing your own operation. You can help another thread complete its unfinished operation, you can help abort an unfinished operation or you can do some sort of waiting. This managed waiting approach allows the use of priorities in lock-free algorithms; lower priority threads can wait longer. This avoids the problems of priority inversion. Care must be taken so that the contention manager's state itself does not become a worse source of contention.

4.4 Memory reclamation

The lock-free stack implementation in listing 4.1 explicitly deleted the popped node. Without a garbage collector, thread safe memory reclamation must be handled explicitly. As it turns out, this is a very difficult problem, especially when it has to be lock-free. The problem in the stack example is that even if you have removed a node from a collection, someone else might be holding a copy of the pointer value (a.k.a. dangling pointer). If you simply delete the pointer and another thread uses this pointer the program might crash. Line 28 in listing 4.1 is an example of a dangerous pointer usage.

4.4.1 Naive approach

A naive solution would be to just queue up pointer values for some sufficiently long time and then delete them. But in general there is no sufficiently long time that is safe, thread might be swapped out for long periods of time. But what if there was some OS specific (non-portable) detail which provided a maximum wait time for swapped out threads? Well, the memory of the allocated nodes during that period might very well exceeded the total physical memory. This would cause the program to run out of memory or start swapping memory to disk.

4.4.2 Quiescence periods

Another approach is to rely on *quiescent periods*, that is periods when no threads are accessing a particular concurrent object. If the existence and frequency of quiescence problems can be guaranteed, the memory can be reclaimed during these periods. Guaranteeing how a concurrent object will be accessed at runtime can be difficult and limits usability. If all threads accessing the object can somehow be interrupted or temporarily blocked, quiescence can be achieved. In practice, this is often implemented as some kind of "'stop-the-world"' language functionality where all execution is paused and memory can be reclaimed. Of course, while the world is stopped the lock-free progress condition is broken. Long pause times can very well be unacceptable in lock-free implementations (cf. [7]).

4.4.3 Hazard pointers

The hazard pointers [34] solution has received a lot of attention lately. It requires you to state a maximum amount of simultaneous threads, N, that can be reclaiming data. Each thread keeps a small number, K, typically two, of hazardous pointers which are nodes the thread is currently accessing. When a thread wants to reclaim a node, it puts it in its thread specific garbage list. Whenever 2*N*K nodes have been collected, it removes all of them from the garbage list and tries to reclaim them. Before reclaiming, it reinserts all nodes marked as hazardous (by other threads' hazard pointers) in the garbage list and aborts those reclamations. The remaining nodes are reclaimed. At the very least, (N+1)*K nodes are reclaimed every turn. The reclamation step can be implemented so it runs in linear time which gives an amortized constant cost per reclaimed node.

Hazard pointers is not truly scalable since the amount of unclaimed data at any given time is $O(N^2)$. This might not be a problem until the number of simultaneous threads become really large. 1000 threads with K=2 has an unlikely theoretical maximum of about 2,000,000 unclaimed nodes at any given time. This would be in the order of 40Mb of data, which is manageable.

An example hazard pointer implementation, which only uses point-to-point synchronization, is available in the second part of this thesis(§13.3).

4.4.4 Reference counting

A common approach to ordinary garbage collection is *reference counting*. All references which refer to the same object share a common counter which keep track how many references there are. Every time a new reference is created the counter is incremented and when a reference goes out of scope, the counter is decreased. When the counter reaches zero, the reference can be reclaimed.

The counting implementation needs to be thread-safe and this can be rather expensive. It also needs to be lock-free if we are to use it in lock-free objects and this complicate things further. Furthermore, a slow thread might keep an entire chain of references alive by holding a single reference, which in turn holds another reference, and so on. Several implementations require double word CAS which is expensive and not commonly available in contemporary processors.

A comparison between a quiescent approach, hazard pointers and reference counting is attempted in [22]. [15] combines hazard pointers with reference counting to get the best properties of both approaches.

4.4.5 Read-copy-update

A fast, but typically blocking, reclamation scheme is *read-copy-update* (RCU). It keeps track of everybody who is accessing pointers, rather similar to reference counting. When a shared pointer shall be reclaimed it waits until all who are currently holding copies cease using them. It then reclaims the data and continues execution.

4.4.6 Automatic garbage collectors

At first glance languages with automatic garbage collectors automagically solve the memory reclamation problem. Thinking a little closer about it, garbage collectors themselves face the same problem as a hand-made implementation. The problem might very well be worse since you do not have explicit control over how garbage collection is carried out. Even if you can configure the garbage collector, it might not be good enough for a heavily contented lock-free container. Making matters worse, you typically cannot control how a garbage collector behaves in just one part of a program.

On the other hand, garbage collectors can have more control than hand-written memory management as they are typically a part of the language. For instance they might be able to safely pause all threads (a.k.a. stop-the-world), which generates a quiescent period where data can be reclaimed. But, when extreme amounts of node data are constantly being allocated, your application might spend most of it times starting and stopping threads.

4.5 Transactional memory

Lock-free objects can provide an extremely efficient form of shared memory communication. But real world lock-free programming is close to rocket science and

lock-free components do not extend and compose well. Another rapidly emerging synchronization approach is *transactional memory*. Some implementations have provided fair performance, is easy to program and composes well. Basically, the code you want to execute atomically is just enclosed in an atomic block and transactional memory system does the rest for you. Sounds like magic? Let us see.

4.5.1 Optimistic execution

Transactional memory work much like database transactions. A common strategy is that all code in an atomic statement is executed *optimistically*. A log keeps track of the performed shared memory operations. After the execution completes, all logged reads are checked so that the values have not been updated by other transactions. The check can be implemented by having a timestamp associated with each shared memory location. If a collision has occurred, the write operations in the log are rolled back and the transaction is retried. If all reads were valid, all the steps of the transactions are committed. The collision check and the entire commit are somehow executed as a single atomic operation. The transactional memory system implements this atomicity for us.

Optimistic execution is efficient but can make running transaction see the effect of other half-finished transactions (see §4.5.3). There are proposed variations which provide better isolation guarantees. This can be achieved by not using optimistic writes but rather doing them in the commit phase or by aborting a transaction immediately as soon as a dangerous value is read.

4.5.2 Example code

Let us revisit the pop method from the stack example in §4.3.3. The lock-free implementation looked like:

```
new_node->item_ = item;
do {
  new_node->next_ = head_->next_;
} while (!compare_and_swap(head->next_,
                           new_node->next_,
                           new_node));
```

A transactional memory equivalent would be:

```
new_node->item_ = item;
atomic {
  node->next_ = head_->next_;
  head->next_ = new_node;
}
```

The CAS and do while-loop is very similar to transactional memory optimistic retry-until-success approach but you do not have to write it yourself.

4.5.3 Side effects and dangers

Transactional memory can be a bit dangerous. If transaction execution is done optimistically, broken invariants caused by other half-finished transactions can be observed. A transaction which observes broken invariants will be rolled back and retried once it completes. But what if it crashes or does not complete? Table 4.1, an example from [21], demonstrates the danger.

Transaction A	Transaction B

`atomic {`
`  if (x != y)`
`  while (true) { }`
`}`
	`atomic {`
	`  x++;`
	`  y++;`
	`}`

Table 4.1: Example of dangerous transactions

There is an assumed variant that x is always equal to y. Transaction A gets stuck in an eternal loop if it sees a half of transaction B.

Another danger is side effects. The transactional system cannot redo side effects, other than writes to shared memory. Therefore, you should never perform any I/O inside transactions. Consider table 4.2, which show two transactions running in parallel with the incrementing transaction of the previous example.

Dangerous transaction	Safe transaction

`atomic {`
`  if (x != y) launch_missiles();`
`}`
	`bool launch;`
	`atomic {`
	`  launch = (x != y);`
	`}`
	`if (launch) launch_missiles();`

Table 4.2: Why I/O should not be performed in transactions

4.5.4 Waiting mechanisms

[51] suggested a composable mechanism for waiting could be added to transactional memory implementations. The idea is to add two keywords, *retry* and *orElse* which can be used inside transactions.

Retry

Retry is meant to replace condition variables. It lets the transactional memory implementation wake the threads which are waiting for a particular condition. Consider the following transaction:

```
atomic {
  y = 5; // Will be undone
  if (x == 10) do_stuff();
  else retry;
}
```

If x is not 10, the transaction will be rolled back and the thread will go to sleep. The transactional memory systems figures out that the thread is waiting for x to change. When another transaction alters x, the thread is awakened and the transaction is retried. This is similar to condition variables but has some notable advantages; The programmer does not need to implement waiting and notifying explicitly and lost wakeups can never occur (see §3.2.4). There is no need to figure out a complex condition first and *then* do shared memory updates. The updates can be done optimistically as you go along and are automatically rolled back if needed.

orElse

Traditional *choice* implementations like POSIX's select or Window's WaitForMultipleObjects are rather cumbersome to work with. They require you to collect all possible conditions that can occur, then wait for one of them to occur in a single place. This can turn the structure of your program inside out.

orElse allows transactions to be composed arbitrarily:

```
void a_transaction() {
  atomic {
    if (x == 10) do_stuff();
    else retry;
  }
}

void another_transaction() {
  atomic {
    if (y != 20) do_other_stuff();
    else retry;
  }
}

void composite_transaction () {
  atomic {
    initial_stuff():
    a_transaction();
    orElse another_transaction();
    else retry;
  }
}
```

If a_transaction retries, it is rolled back and another_transaction is attempted. If another_transaction also retries it is rolled back too. Finally, the composite transaction is rolled back and initial_stuff is undone. The thread executing the composite transaction goes to sleep until x or y is updated by another transaction.

Choice of a dynamic number of transactions

One problem with orElse is that it is a static construct. This makes it difficult to wait on a dynamic number of transactions. It is possible to achieve "'dynamic choice"' by using recursive functions. Assuming have a functor construct that can represent higher order functions, a C++ solution might look something like this:

```
template<class TransactionIterator>
void dynamic_or_else(TransactionIterator begin,
                     TransactionIterator end) {
  atomic {
    if (begin == end) retry;
    function<void()> transaction = *begin;
    transaction();
    orElse dynamic_or_else(++begin, end);
  }
}
```

But a recursive approach is in general not safe in imperative languages, since threads typically have a fix maximum stack size and stack depth. Therefore, it seems likely that some other construct will be supplied which allow dynamic choice.

4.5.5 Cost and design space

Transactional memory synchronization has some overhead but seems to scale really well [21] [51]. It is difficult to give exact numbers on how much transactional memory implementation costs compared to other synchronization schemes. Transactional memory can be optimized for few or many collisions, for fast log checking and commit or for fast undoing, etc. There is also a design space to consider. For instance, should nested transaction be allowed? Should dynamic allocation and memory reclamation be allowed inside transactions? What isolation guarantees should be provided between different transactions? Should shared memory used in transactions be accessible outside of transactions? Should the implementation be lock-free or blocking (in the absence of retry and orElse statements)? These design choices all affect the cost and complexity of the implementation. It is currently not clear if an acceptable transactional memory implementation can be made performant enough for mainstream imperative languages.

The idea of transactional memory has been around a long time. Hardware support for transactions was already suggested in 1986 [28]. To date, no commercial processor supports hardware transactional memory (HTM). Sun announced that its upcoming Rock processor (see §6.4.6) will support some level of transactional memory. Naturally, some parts of a transaction memory system can be implemented in hardware and others in software. Software transactional memory (STM) has been implemented in Haskell (see §6.2.8) but none of the major imperative languages.

What transactional memory does to synchronization has been likened to what garbage collection does to memory management [19]. There is some inherent cost and you lose a lot of control. But you gain a lot of simplicity.

4.5.6 Revisiting threads and locks problems

Assuming transactional memory becomes efficient and widespread, which of our thread and locks related problems (see chapter 3) will be solved? Deadlock, composability, race conditions and lost wake ups have definitely been fixed. The problem of indeterminism is lessened because far fewer and stranger errors are expected to appear. Lack of memory isolation is not fixed. Finally, fix-grain synchronization has been amended which is a key to scalability.

So transactional memory fixes most of the "'difficult to program problems"'. To be fair we have introduced at least two new problems. Lack of inter-transaction isolation if optimistic execution is used and having to delay I/O operations until after transaction execution (see §4.5.3). But these problems seem to be a lot easier to deal with.

4.6 Thread level task scheduling

Another key problem in achieving task level scalability is having a lightweight task abstraction. Transactional memory does not address the fix-grain task level parallelism problem (see §3.4.2). TLP heavy programs should try to run as many tasks as there are available cores. To extract fine-grain task level parallelism, the per task overhead must be small. Since threads are so heavy-weight (see §3.3.1) we have to reuse them to get good performance.

The basic idea of a *thread pool* is to maintain a pool of worker threads, approximately the number of available cores, which perform computation. Threads pools are typically not used for asynchronous requests, prioritizing work or when there is real time constraints involved. The idea is to keep all cores working as much as possible to get maximum computational performance.

4.6.1 Task decomposition

For certain problems, task decomposition can be done before executing any tasks. Other algorithms, such as divide-and-conquer ones, chops up problems into sub problems as tasks are being executed. Task decomposition can also depend on runtime branching within running tasks. Therefore a typical thread pool implementation will allow tasks to submit child tasks. To maintain locality of reference, it is advantageous if the same core or thread executes its own child tasks.

4.6.2 Work stealing

When a worker thread runs out of child tasks it does not need to idle. Instead it can steal work from the other worker threads. Each thread can maintain a concurrent work queue accessible from other worker threads. By putting all of its generated child tasks in its own work queue it can maintain locality of reference as long as stealing does not occur. Naturally, the synchronization cost that stealing incurs must be considered before applying such an approach.

4.6.3 Problems

While thread pools have successfully been used to extract parallel performance (see examples §6.2.4, §6.2.5, §6.2.6 and §6.2.7) they all suffer from a number of inherent problems.

The programmer must perform task decomposition explicitly which requires manual work. In general, the programmer cannot estimate the per task or synchronization overheads in the thread pool implementation. This makes it difficult to know how fine-grain task decomposition should be made. The thread pool scheduler typically does not know how much work each task consists of either, which prevents effective up front scheduling. If the scheduler had somehow done the task decomposition, it could adapt the task size itself to fit scheduling.

More troubling is the problem of double scheduling in desktop machines. The thread pool schedules tasks onto worker threads and the OS schedules threads onto cores. Basically the thread pool scheduler is second guessing what the OS scheduler is doing and can only hint at which processor a thread should run at[5]. Neither of the schedulers have all interesting information. The thread pool would like to run as many worker threads as there are available cores at any given moment. But cores can be dynamically given and removed to a certain process. If the OS scheduler knew roughly how much work or many tasks there are left, it could make more long term scheduling decisions. For more information on the scheduling problem, see §5.2.

Impure code (5.4.1) and thread affinity, that code needs to be executed in a particular thread, make it difficult to wrap up some code pieces as independent tasks which can be executed by any worker thread. Exceptions must also be propagated from worker threads to user threads somehow. Furthermore, typical thread pool implementations do not consider memory access patterns or heterogeneous hardware. Still, thread pool implementations can extract some parallel performance, without altering the rigid software stack of process level threading on top of OS level scheduling. As such, it should be considered a success. And better, when OSes start to change, the thread pool implementations can keep their interfaces.

4.7 Summary

This chapter explored fine-grain locking, atomic operations, lock-free programming, memory reclamation, transactional memory and thread pools. They are all techniques to extract parallel performance without removing the threading model or shared memory model. The idea is to extract as much performance as possible without changing programming languages, tools and operating systems too much. This is called threading for performance.

Fine-grained locking is very difficult to get right and is not composable. Even

[5] Each thread and process can give a hint as to which processor they would like to be run on. This is called *processor affinity* and can be used to try and maintain locality of reference (see §2.3.2).

so it changes nothing and allows a lot of parallelism to be extracted. Atomic operations are already widespread and provide a faster and safer alternative to locking - as long as you only need to operate on a single memory location at a time. If you need to alter multiple memory locations atomically, which you most often do, you need to consider the memory model of your programming language. This tells you what reorderings are allowed. Lock-free programming builds on atomic operations and memory models to provide really, really fast concurrent algorithms. The downside to lock-free programming is that it is basically rocket science, forces you to think about memory reclamation and does not compose or extend well. Transactional memory also allows executing atomic operations on multiple memory locations. It attempts to give a fair trade-off between speed and simplicity. Transactional memory implies the largest change and could replace locks, conditional variables and perhaps even atomic operations and memory models. It is still being researched and has to date not been implemented in any of the major imperative languages. Thread pools tackle the enormously difficult problem of portable parallel scheduling. They manage to extract some performance without altering hardly anything. The only thing required is some simple model or queries of how the hardware looks, for instance how many cores there are.

Chapter 5

Desired programming models

Even if transactional memory and thread pool implementations become efficient and widespread, huge amounts of parallel performance would still not be extractable. Threading-based implementations can probably provide a good platform to explore future generic parallel programming models even if they cannot extract certain kinds of parallelism efficiently. Domain specific DLP programming models might become more and more generalized and explore the design space from the less generic ends of the spectrum. Meanwhile rearchers will continue to analyze and predict what programming models, which are not compatible with today's OSes and languages, might look like.

This chapter explores some of the vast complexities of constructing fairly generic parallel programming models. Having a basic understanding of such complexities is material to know what types of parallelism a particular programming model can hope to extract.

5.1 Dwarfs

Parallel programming models have been researched for decades. Still, we are nowhere near a set of general purpose models which can extract the different types of parallelism we know of. A big problem is knowing what type of parallel applications there are and what future hardware will look like. To tackle this problem a research group at Berkley has proposed using a number of *dwarfs*[Asanovic:EECS-2006-183]. Each dwarf is a high-level description of a problem and tries to "'capture a pattern of computation and communication common to a class of important applications"'. A hardware architecture can be evaluated on how fast it can compute each of the dwarfs.

A programming model can be evaluated on the number of dwarfs it can express. The dwarf should be expressible in a concise manner without throwing away information (such as data access patterns) needed for efficient parallel

execution. Certain dwarfs require certain information. Expressing all kinds of possibly interesting information in a programming model would lead to a very cumbersome way of programming. It is more likely that a set of programming models will be used, which are concise for the types of dwarfs they are targeting. This will raise the need of efficient and reliable ways of composing different programming models.

5.2 Runtime scheduling and its complexity

Perhaps the most central and important problem in extracting parallelism is that of scheduling. How do we map tasks to cores in time and space? How do we optimize data flow and intercore communication? How do we efficiently execute programs from some programming model on various CMP architectures? Contemporary desktop operating systems typically perform simple task level scheduling. This is a poor approach for scheduling DLP and synchronization intense programs onto CMP architectures. Certain well isolated hardware resources such as the GPU and its dedicated memory is scheduled separately. Desktop programmers can also layer their own schedulers on top of the OS scheduler (see §4.6).

This section explores some of the complexities a future desktop manycore scheduler must face and, in particular, what information it will be interested in. It is assumed there is one central scheduler which has a fairly accurate model of the underlying hardware; Number of cores, how many memory banks, how cores and memory banks are interconnected, what latencies and bandwidths there are, etc. It also assumes that different types of parallel programs are running concurrently.

5.2.1 Task level scheduling

A good starting point when analyzing scheduling is task level parallelism. The scheduler has a number of tasks, hopefully more than the number of cores, which it maps in time onto cores.

For some simpler computational problems, all subtasks are known directly. More commonly, task submits new child tasks as they are being executed. The scheduler might not know how long time a task will take. Longer tasks can be time sliced. A more optimal task level scheduler might even be able to split certain programs into suitably fine-grained tasks itself.

5.2.2 Fairness and priority

Scheduling is not just about maximum hardware utilization. Certain tasks want to finish fast even if it means suboptimally consuming a lot of hardware resources. Prioritizing fine-grain tasks individually is not optimal. When some piece of work is broken down into finer grain tasks, there is rarely any need to finish an individual sub task. It is not until all subtasks are completed that some

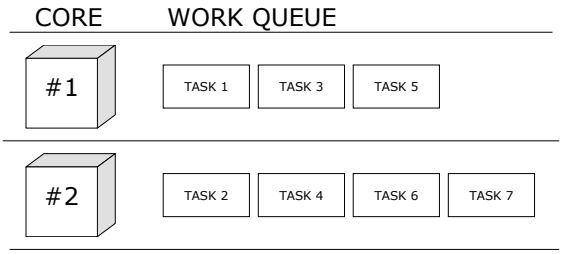

Figure 5.1: Task level scheduling

value is achieved. In such scenarios, the set of subtasks should be prioritized together to achieve minimum total computation time. Some programs or tasks might also have higher priorities than others. Some degree of fairness might also be required to avoid starvation.

5.2.3 Heterogeneous resources

Some cores specialize in floating point or integer arithmetic, some in logic or branching. Other simpler cores might form a large grid suitable for highly parallel applications. Some cores might be very domain specific and customized for areas such as 3D-rendering or physics simulation. If heterogeneous cores share a lot of resources, perhaps even memory, they benefit from being scheduled together.

Scheduling heterogeneous cores together is naturally a more difficult problem. The scheduler must at least know what type of core different tasks wants to be executed on, perhaps by using hinting (§5.5.5). Specialized cores might have their own instruction set[1]. Programs that can be executed on any core will then need some virtual machine or compilation step at loadtime or runtime. Alternatively, each core can execute the same instruction set but specialized cores execute certain instructions faster. This is called single instruction set architecture heterogeneous CMPs [29].

5.2.4 Intertask communication

Almost all tasks need to synchronize with each other. This might be explicit access to shared memory locations which other cores are also accessing. Or it can be returning a value to the parent task. When the cost of intertask synchronization becomes comparable to the computational work per task, you

[1]Intel and AMD, the two largest desktop processor manufacturers, have quite different strategies on this point [13] [52].

will get less and less performance the more cores you *scale out* a problem to. At some point the computation will actually run slower, the more a problem is decomposed and scaled out. To allow better scaling, a manycore processor could be equipped with a fast, low latency and low bandwidth interconnect use for broadcasting small synchronization messages [1].

A manycore scheduler will be interested in knowing how much synchronization will occur between tasks. If this information is available, the scheduler can directly select how many cores a particular problem should be scaled out to. If it is not available, the scheduler and hardware may be able to detect various kinds of contention and adaptively scale out a problem until the performance increase flattens out. Of course, that requires that the scheduler can somehow *measure* contention or preferably performance.

5.2.5 Data level scheduling

Certain problems involve really simple computations but massive amounts of data. In such data level parallelism intense problems, the problem at hand is setting up efficient data *streams*. For DLP problems it is better to schedule data flow to memory interconnects than tasks to cores. If memory bandwidth is the limitation, a particular problem should not be scaled out more than bandwidth allows.

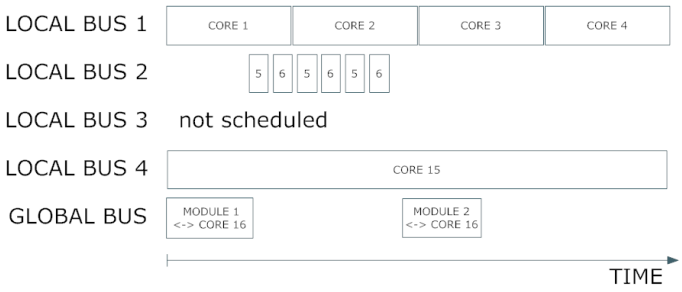

Figure 5.2: Example scheduling of the NUMA processor in figure 2.5

In realistic manycore processors, the situation will be more complicated than in figure 5.2. Memory bandwidth and latency between different cores and memory modules will differ, interconnection schemes will be more complicated, there can be heterogeneous cores, etc. A well performing scheduler should have a fairly accurate software model of this and schedule operations onto this model.

Caching is of little help when large data amounts is being processed since new non-cached data is constantly being accessed all the time. Therefore each small piece of memory that is accessed will have to pay the latency cost of a cache miss. If it is known in advance what data a particular program will need, that data can be scheduled to arrive just when it is needed or copied to a fast on-chip memory shortly before. This is called *prefetching* data. When huge

data amounts are constantly being fetched and processed on the fly, it is called streaming.

5.2.6 Sum of complexities

Most parallel problems are limited by either computational power, synchronization latency or memory bandwidth. Some problems are limited by more than one of these factors. But on a desktop CMP, executions of all these types of programs need to be scheduled together. A parallel scheduler must have a fairly accurate model of the underlying hardware to make a good job. It must be also handle priority, fairness, potential heterogeneous cores and perhaps other shared I/O resources. It might also need to consider power consumption. It is unclear how and if these issues can be separated. Some of the problems may be handled sufficiently well by hardware and simplified away from software layers. Even so, manycore scheduling will be really complex.

5.3 Parallelism's implication on programming models

This section explores some of the most important aspects that parallel programming models must consider.

5.3.1 Programming models as scheduler interfaces

Clearly, we do not want to expose the gory details of scheduling in a programming language. We would also like to hide as much of the hardware as possible, both to simplify programming and to increase portability. No programming model is going to completely succeed in hiding this complexity away. Just like contemporary programmers need to pay attention to locality of reference, future parallel programmers will need to have some mental model of the hardware and perhaps the scheduler. By having a rough idea how a scheduler works, we can understand what information is required to extract parallel performance. Care can then be taken to declare this information in the programming model and preserve it until scheduling.

One, perhaps radical, way to think of a parallel programming model is like an interface to a scheduler. If the programming model is targeting problems limited by synchronization latencies, information about how synchronization occurs must somehow be present in the programming model. Programming models targeting DLP applications must express memory access patterns. They might work with operations on large arrays of data. Others, so called data flow or reactive programming approaches, focus on how data flows through a system through simpler computational elements. If TLP is targeted, information on how tasks are, or can be, decomposed must be present. But if you require the programmer to state too much information, the programming languages becomes cumbersome to write, to understand and to maintain.

The information required to efficiently schedule and execute the code must be present in the programming model. Some information can be used to schedule or decompose tasks at compile time. But for desktop programs, you want the same compiled program to run on various CMP and be scheduled according to dynamically varying system resources. This requires a lot of information to be preserved until loadtime or runtime. It seems likely that the boundary between compile time and runtime will blur and that virtualization will become even more important thanks to the manycore shift.

5.3.2 Explicit vs. declarative

Of course, the programming model is also a interface to human programmers and must not become too abstract or complicated. It should be explicit enough to express effective algorithms and implicit enough to be simple and reusable. Many of the problems with threads and locks are due to overly explicitness. The programmer explicitly decides how many threads of execution there shall be, explicitly divides work into tasks division and explicitly performs synchronization. If we want to abstract away hardware details and scheduling implementations we clearly cannot be that specific.

A piece of code is *declarative* if it declares what something is like rather than specifying how it should be executed. GUI code can declare that there is an ok button in some dialog rather than explicitly drawing the button. The drawing is left some library which knows how and where the button should be drawn. The dialog code becomes simpler and more concise. If you want to change button look you can typically do it in a single place rather than having to change the code of all dialogs. On the other hand, if you want change the look of single specific button, the GUI library might not be expressive enough to support it. There is often a trade-off between expressiveness and conciseness.

Explicitly stating how many threads should be running is not a good idea when writing scalable and portable code. It might be sweet to not have to express task decomposition explicitly either, in some scenarios the compiler or runtime could figure this out for you. And would it not be nice if they could also worry about synchronization for you? But all of a sudden, you cannot express your really smart and efficient shared memory algorithm anymore.

5.3.3 Understandable

There is another danger in declarative approaches. If the underlying implementation of a programming model becomes too complex, programmers might not *understand* why a certain piece of code executes rapidly or slow. If a programming model's transformation from code to scheduled execution is like a black box, it will be impossible to profile a program and figure out what is going wrong. And performance is what motivated the manycore shift in the first place. In [1], it is predicted that profiling will need to become a first class citizen. Hardware, operating systems and virtual machines will need to supply well designed APIs which provides useful metrics such as cache misses, page

faults, thread swapping, per core load, task buffer sizes, memory bus collisions, etc.

Both schedulers and parallel algorithms are going to be littered with parameters and settings. Just consider all the parameters uncovered in the few examples of §2.3 and §2.4. Tuning these parameters is going to be time-consuming but critical to achieving good performance. Many of the parameters will depend on the hardware. Others on what system resources are dynamically available. [1] suggests there should be *auto-tuners*, which help search through the vast parameter space and determine good settings. Care must be taken to make any automation understandable and predictable, otherwise the programming model risks turning into the black box described above.

5.3.4 Debugging and fault tolerance

Debugging tools will also need to be adapted to parallel programming models. Many bugs will only appear as a result of complex interaction between multiple cores. Such situations must be understandable if the bugs are to be fixed. Programming models should try to disallow expressing tricky and indeterministic bugs. For instance, it is easy to imagine programming models which does not have dead-locks or race conditions (see §4.5 for an example). At the very least, it should be not be cumbersome to reason about correctness and make proofs.

As more transistors are added and they become smaller and smaller, errors will become more common. A *soft* error is a onetime error, for instance caused by an alpha particle or tunneling effects. A *hard* error is an error due to a broken component. Future CMP will face higher soft and hard error rates [1]. Many of the errors can be detected and handled, by having some amount of redundancy, in hardware. Some level of fault detection and recovery might be needed in the implementation of programming models. Hopefully, processor and memory errors do not need to be exposed top-level in actual programming models.

5.3.5 Composability

Modularity is and will always be extremely important in software development. We want to break down software into small pieces, test and verify them independently and then compose them into software layers and programs. We do not want big monolithic software components as those are difficult to both build and change.

Composability is applicable at many different levels of parallel software. You want hardware instructions executed at one core to compose atomically with instructions executed on another core, even if they address the same memory location. You want to see shared memory modifications take place in the same order for all cores, consistent with the program order of all cores(see §4.2.4). You want to allow multiple low-level atomic operations to be combined into a larger composed operation, which is still executed indivisibly. You want to allow atomic operations involving components from both library X and library Y. You want progress conditions (see §4.3.1) preserved through the entire software

stack. You want different top-level programming models to be compose well with each other. And you cannot afford to lose too much performance in every composition step. That would result in a multiplying effect and make highly composed top-level code very slow.

Composing parallel software is a lot more difficult than sequential. Small subtle details in seemingly identical definitions can have vast impacts on composability. A good starting point in understanding parallel composability is to get a deep understanding of linearizability and sequential consistency (see chapter 8).

Composability should be a top concern when designing parallel code. For instance, the concept of a lock seems like a great and intuitive abstraction but is considered flawed because locks compose so poorly (see §3.2.2). Another example is composing two data intense parallel algorithms or programming models. To do this efficiently, you might need to give up explicit data structures and let the programming model handle data layout and flow [1].

5.3.6 Today's broken software stack

Much of today's desktop software stack (drivers, OS, languages, compilers, virtual machines, libraries, tools) is not well adapted to parallelism. Historically, parallelism has not been very important, so all layers have been cheating a little bit. As an example, some processor manufacturers had very weak or vague memory consistency guarantees(see §4.2.4) until around 2006. This lead to confusion and bugs higher up in the software stack, even in an as renowned compiler as gcc [4]. Java's concurrency specification was impossible to implement on many common processors until it was updated in 2005 [43] [26].

Even with these problems fixed, the desktop software stack is still adapted to threading and a uniform shared memory model (UMA). The OS kernel monopolizes on CPU scheduling and only performs basic task level scheduling without full information (see §4.6.3). Memory access is not scheduled at all. Iteratively adding new parallel programming models to this existing framework is going to be a real challenge, to say the least. Virtualization at appropriate levels of the existing software stack might allow new models to co-evolve with the mainstream ones. You might even provide a specialized legacy core, which executes older instruction sets fast [1].

5.4 Some interesting properties

This section introduces a few programming model properties, which are going to be increasingly important in a parallel setting.

5.4.1 Pure

Functions which *do not access hidden state and have no side effects* are said to be *pure*. A pure function does not perform any I/O, such as writing to shared

memory, and does not depend on any mutable state. The result only depends on its input parameters. You can also call it as many times as you want, as it will not affect anything.

Purity is a very nice property when analyzing code correctness and when performing compiler optimizations. It helps separate I/O and pure computation and logic. It is even more interesting in a parallel setting as pure functions can be parallelized behind the scenes - it does not matter by who or when they are executed, the result is the same. Of course, the input data to the function needs to be transported to the core executing it and the result must be delivered back.

5.4.2 Immutable

Immutable (a.k.a. read-only) data cannot be modified. Multiple cores can read such data simultaneously without worrying about synchronization since nobody is altering the data. Immutability is also an important property for reasoning about code correctness.

Definite assignment is another similar concept. Definite assignment data can only be assigned to once, after which it becomes immutable. The concept could be extended to any data which can be frozen. Arbitrary mutations can occur but at some point, the data freezes and becomes immutable.

Synchronization can still be needed if the immutable data needs to be reclaimed during application lifetime.

5.4.3 Value semantics

Data is semantically passed *by value* or *by reference*. If a program alters data sent with value semantics, the original data is not updated. This does not necessarily mean that data must be copied, references can be used behind the scenes. Reference semantics can be very useful, but can also lead to a world of problems. Dangling pointers, unclarity who can access what data, unexpected data flow and state changes, poor locality of reference (see §2.3.2) etc.

In parallel programs, accesses through references must be properly synchronized. This can be difficult to implement and prove correct. Furthermore, compilers and runtime schedulers get a very difficult problem of predicting synchronization and memory access patterns when references are involved. Yet another problem in NUMA architectures is that references, which are passed between cores, can be very expensive to access.

One can distinguish between *local* and *global* data where local data belongs to a particular *node* (processor, core, thread, etc). The parallel problems of references can be avoided by only using references to local data. These reference should never be shared across nodes. I.e. you think of data as belonging to a node, even in a shared memory system. Message passing, producer-consumer or publisher-subscriber components can be used for internode communication.

If the programming model clearly distinguishes all internode communication, this information can be passed along the scheduler which can then do a better job.

5.4.4 Move semantics

A problem with reference-less communication is that big data structures can be expensive to copy across nodes. If there is globally fast and shared memory available, it should be used. A producing node can allocate global memory, work with it a while and then publish an immutable data structure. Or *move semantics* can be used where one node gives up ownership of some data and transfers it to another node. Erlang (see §6.3.5), Singularity (see §6.3.6) and C++0x(see §6.2.2) all include some notion move semantics or publishing.

5.5 Automatic extraction of parallelism

Ideally we would like programmers to bother as little as possible with parallelization details. Granted, the programmer must supply the required information for efficient scheduling (see §5.3.1). But once that is done, it would be terrific if the programming model could perform optimal task decomposition, figure out the best synchronization patterns, etc.

Unfortunately, it is not enough to theorize about what information is needed. You cannot throw an infinite amount of complexity at the compiler and runtime scheduler and expect them to automatically extract all parallelism and perform optimal scheduling. Both compilers and contemporary OS kernels, which perform thread-level scheduling, are already very complex beasts.

5.5.1 Computation as a directed acyclic graph

Which parts of a program can be executed in parallel? The answer is; it *depends*. It depends on which parts of a program requires *data* from previous computation. It depends on the *control flow*, that is how execution flow is altered by if-statements, for-loops, virtual function calls etc. It also depends on *input and output* operations. If you change the order in which a program inputs data or outputs it, you change program behavior.

Consider the following code:

```
1  int a = task_a();
2  int b = task_b();
3  int c = task_c(a);
4  if (b > 123) {
5    output(c);
6  }
7  output(a);
```

Listing 5.1: Task parallelization example

Which parts of this program can be executed in parallel? Assume that task a, b and c are pure (see §5.4.1). One way to analyze the potential parallelism is to draw a directed acyclic graph (DAG) of all data, control and I/O dependencies between tasks.

Analysis of the DAG in figure 5.3 leads to the conclusion that task a and c can be executed in parallel with b and the if-statement. No other parallelization is

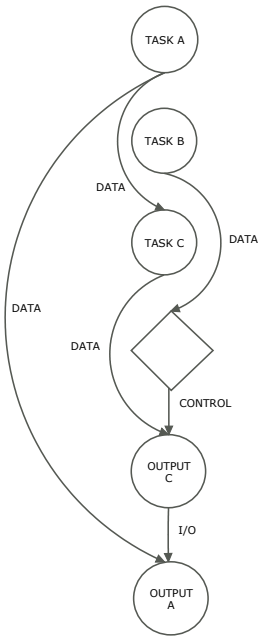

Figure 5.3: Dependencies between tasks of example 5.1

possible with this task decomposition. A more precise methodology on analyzing and extracting compile time task level parallelism can be found in [17].

This type of reasoning can be applied to every level in a program; from hardware instructions (ILP) to distinct steps in an algorithm or even top-level program structure. The more fine-grain tasks it is applied to, the more you have to worry about of the cost task abstraction and synchronization. Advanced task level schedulers (see §5.2.1) can use DAGs to make scheduling decisions. DAGs can also be used to determine how parallelizable different algorithms are, which can be helpful when selecting an algorithm.

5.5.2 Speculative execution

Speculative execution is executing code whose result might not be needed. Rather than waiting for data required to make a branch decision, one of the control branches can be speculatively executed. If the decision was wrong, the execution is thrown away and resumed at the branch. Needless to say, non-pure code should not be speculatively executed.

If large parts of a program are pure, more parallelism can be extracted because there are less I/O dependencies. Larger parts of the code can also be speculatively executed, since the side effects are lumped together in certain parts of the code.

Another interesting prospect with parallelism is executed *both* branches. If there are a lot of cores sitting around doing nothing, they might as well execute code. As always, the price of general complexity and synchronization must be considered. This scheme of using idle cores can also be employed top-level. Consider two algorithms, one with a fast average but unacceptable worst case behavior, another with slow average but acceptable worst case behavior. Both algorithms can be executed in parallel. As soon as one finishes, the other algorithm can be aborted [40].

5.5.3 Extracting fine-grain task level parallelism

In theory, most programs exhibit a lot of fine-grain task level parallelism, which can be automatically extracted at compile time. Especially pure (see §5.4.1) parts of a program which only needs to consider data and control dependencies. But the cost of synchronization, for instance by expensive atomic instructions or memory fences, might be greater than the speedup from achieved parallelism.

Another problem, which turned out to be a real showstopper [37], is compilation time. Research indicates that it takes a lot of time to analyze code at compile time and figure out which parts of a program can be parallelized. In real world projects, compilation time is a really precious commodity. You ideally want an entire system built, integrated and tested in a manner of minutes - every time you submit code to the version control system[2].

Basically, the finest grained parallelism seems difficult to harvest. It is better to aim for a bit larger tasks. Unfortunately, with contemporary programming models, most of the task decomposition burden falls on the programmer.

5.5.4 Domain specific applications

Some domain specific programming models have been very successful in extracting automatic parallelism. Structured Query Language (SQL), a declarative approach to extracting information from data bases, is probably the best known. You do not explicitly state how a query shall be carried out, you rather declare what information you want. 3D-rendering, especially for games, is a successful and highly automated data level parallel domain for desktop systems.

5.5.5 Hinting

There is a grey zone between explicitly stating how something is parallelized and not stating anything about it at all. SQL queries can be structured in certain ways because the user *knows* how the database implementation will execute them [27]. Naturally, relying on implicit assumptions is dangerous. A programmer might not understand why a certain query is written in a certain way and might break a performance critical detail when altering it. Comments have to be used to convey implicit information, but are often written and read sloppily and have a tendency to drift out of sync with the code.

[2]This is generally referred to as *continuous integration*.

A trade-off between explicitly stating too many parallelization details in APIs and not doing it at all is *hinting*. The programmer can optionally pass on some parameters, which states how the implementation *might* execute the code. For instance, hinting could be used to specify that a particular task makes heavy use of floating point operations. A task level scheduler can then run the task on a floating point core, if one is available.

5.6 Embedded computing

Although the focus of this thesis is on desktop parallelism, the problems in embedded domains are pretty much the same. Embedded applications often have hard real-time constraints, which will affect programming model trade-offs. They typically require more reliable software, which might be argument enough to stay away from indeterministic programming models. Finally, embedded computing target somewhat different domains. Other than this, embedded and desktop computing share practically all problems of the manycore shift. It seems likely that the desktop and embedded industry will start working closer together.

5.7 The challenge

The challenge at hand is to deliver scalable, portable performance for the mainstream programmer. Many application areas, which have been too computationally demanding, will open up to desktop computers. These application areas must be the ones driving sales of both hardware and software. Nobody wants to pay for theoretical performance. Likewise, new applications require existing hardware and lower software layers. Hardware, compilers, operating systems, middleware and applications will need to co-evolve to deliver performance. The historically quite separate hardware and software industry will need to work together to solve this chicken-and-egg problem. Highly data parallel "'low hanging fruits"' [1] will be the first application areas where parallel performance is harvested. Threading for performance (see chapter 4) will deliver some task level performance.

Mainstream programmers cannot become parallel experts over night, nor can the desktop software stack change very fast. DSLs, libraries, middleware and virtual machines will become increasingly important in extracting parallel performance. Virtualization at suitable points in the software stack can allow new programming models and tools to co-exist with existing ones. Certainly, there is a lot of interesting and ground shaking movement in both the hardware and especially the software industry. The next chapter surveys some of the parallel projects that the industry is undertaking.

Chapter 6

The contemporary industry

This chapter is not finished yet.

What is the industry doing to deliver parallel performance to the mainstream programmer? Quite a lot.

6.1 Contemporary programming models

Herb Sutter described what is going on in the software industry right now as a land rush, similar to what occurred when GUIs and object orientation became mainstream [48]. There are a lot of projects which aim to deliver parallel performance and this sections list some of them. It is not a comprehensive list, rather a selection of some interesting projects. Evaluations or reviews are not given either.

The programming models are divided into threading for performance projects and non-thread based projects

6.2 Threading for performance models

This section some important software efforts for extracting thread-based parallel performance. Most are aimed at improving existing mainstream shared memory imperative languages.

6.2.1 Java memory model

Desktop CMPs or even dual processors were considered quite esoteric while uniprocessors were predominant. And parallelism was, and is, extremely complex to get right. Both hardware and various software layers only gave parallelism so much effort. In the light of this, it was not strange that Java's memory model (see §4.2.4) specification was flawed. These flaws were fixed in Java 1.5, which was released in 2005. The update involved some change in semantics.

> The Java Memory Model was an ambitious undertaking; it was the first time that a programming language specification attempted to incorporate a memory model which could provide consistent semantics for concurrency across a variety of architectures

[32]

6.2.2 C++ 0x

The next version of C++, called C++0x, will include some basic multithreading abstractions and a new memory model. The memory model is inspired by Java's but it is much more advanced, as C++ needs to be a lot more efficient than Java. Some of the same people working on the Java memory model are now working on C++'s. The second part of this thesis explains how this memory model works. C++0x also includes move semantics (see §5.4.4), although the motivation for this was efficiency rather than concurrent correctness.

The next library extension of the language, Technical Report 2, aims to include some new abstractions such as future objects and a thread pool specification (see §4.6). The boost community, www.boost.org, is also involved in these and other ambitions.

6.2.3 Channel

Channel is a C++ project, which has the rather ambitious goal to capture the design space of named message passing in a generic library. It provides name spaces for asynchronous, distributed message passing and event dispatching. See http://channel.sourceforge.net/.

6.2.4 OpenMP

OpenMP is a very successful and widely used software specification. It provides an API for portable shared memory parallel programming in C, C++ and Fortran. Most major compilers support the OpenMP API, which is enabled with a compiler flag.

> The directives extend the ... base languages with single program multiple data (SPMD) constructs, tasking constructs, worksharing constructs, and synchronization constructs, and they provide support for sharing and privatizing data.
> The OpenMP API covers only user-directed parallelization, wherein the user explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel. OpenMP-compliant implementations are not required to check for data dependencies, data conflicts, race conditions, or deadlocks, any of which may occur in conforming programs. In addition, compliant implementations are not required to check for code sequences that causes a program to be classified as non-conforming. The user is

> responsible for using OpenMP in his application to produce a conforming program. OpenMP does not cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization.

6.2.5 java.util.concurrency

Java has a rather rich set of locking and non-locking containers in its standard library. There is also a work stealing thread pool, a implementation of a future object and some other abstractions. This library is being expanded. See http://gee.cs.oswego.edu/dl/concurrency-interest/index.html, or a more digested summary at

http://www.infoq.com/news/2007/07/concurrency-java-se-7.

Some task parallelism related parts of java.util.concurrency are also known as the fork-join framework.

6.2.6 .NET

.NET parallel extensions (a.k.a. Parallel FX) is an update for Microsoft's .NET platform which targeting parallel performance. Parallel extensions is quite similar to java.util.concurrency. It includes a Task Parallel Library which employs work stealing and provides a future object, just like Java's fork-join framework. Parallel extensions also supply some parallel-for and parallel-for-each constructs which target very basic thread-based DLP.

LINQ (Language INtegrated Query) is a .NET standard for querying various data sources, such as arrays, XML documents or even relational databases, as long as they have some object representation. The idea is to hide the underlying data representation and provide combined queries on different types of data sources. LINQ's syntax is very similar to SQL6.3.1 and is incorporated into several .NET languages. Parallel extensions supply a parallel LINQ execution engine, called Parallel LINQ (PLINQ). Unfortunately, it also needs to extend LINQ's interface with some explicitly parallel directives. PLINQ targets same type of data level parallelism that SQL databases does.

Microsoft has also been working on a unified memory model for all their native code, called PRISM.

> ... a single memory model for all native code on Microsoft platforms, including the source code, compilers and tools, and supported hardware platforms for Windows XP/Vista (client and server), Windows Live, Windows Mobile (Smartphone and Pocket PC), and Xbox.

[47]

This memory model has co-evolved with C++'s new memory model which is discussed in the second part of thesis.

6.2.7 Intel threading building blocks

Intel's Threading Building Blocks (TBB) is a thread-based open source library for C++. It provides a sophisticated task level scheduler, which can deal with complexities like hot and cold data, locality of reference and basic automatic and manual task decomposition from iterator ranges. It supplies parallel_for and parallel_reduce templates which employ the task scheduler, but direct access to the scheduler is also available.

TBB includes a simple linear data flow programming model through its pipeline abstraction. The pipeline also employs the task scheduler and its worker pool as a backend. It considers performance critical parameters such as cache sizes to scale out a pipeline stage to an optimal number of cores. On top of this it supplies a few lock-free and fine-grain lock-based concurrent containers, a lock-free memory allocator, mutexes, atomic operations and fences. Some of its functionality will be covered by the next C++ standard, C++0x, and much of the remaining functionality is targeted for future C++ standard extensions.

TBB supports Linux, Macintosh, Sun and Windows operating systems. It can be mixed with other threading packages for C++ such as, OpenMP, pthreads or Windows threads. TBB has a well written tutorial which explains a lot of interesting design rationale, algorithms and techniques in the implementation and also general considerations that must be made when "'threading for performance"'. It is available at

http://www.threadingbuildingblocks.org/documentation.php

6.2.8 Haskell

Haskell is a pure (see §5.4.1) functional language which makes it interesting from a parallel point of view. Haskell has support for threading, software transactional memory and also some explicit "'control parallel strategies"'. There are a lot of other suggested parallel extensions, for instance Data Parallel Haskell (DPH). DPH uses a very elegant programming model called *nested data parallelism*, which unlike *flat* DLP approaches is not limited to applying a single computation to all elements in some collection.

6.3 Non-thread based programming models

It is difficult to draw a sharp line between threading for performance and non thread-based programming models. This section lists some notable programming models which are at least not directly related to and focused on threads.

6.3.1 SQL and databases

Structured Query Language (SQL) is perhaps the most successful and widespread approach to parallelism. It is a set-based declarative language for retrieving and inserting data in relational databases which can contain enormous amount of

data. Typical database implementations use some kind of optimistic approach to executing queries which detect data inconsistencies and collisions afterwards.

6.3.2 RapidMind

RapidMind provides a C++ library which uses a *Single Program Multiple Data* (SPMD, a.k.a. flat DLP as opposed to nested DLP) approach where the same program is executed on all the elements in a collection. RapidMind has no language extensions, only macros and types which form an embedded language in which parallelizable parts of a C++ program can be expressed. A basic taste of the API and the example code can be found at:

http://www.rapidmind.net/pdfs/WP_RapidMindPlatform.pdf

RapidMind has three backends, one x86-backend for AMD and Intel multicore CPUs, a GPU-backend for ATI and nVidia GPUs and one for the Cell (see §6.4.3) processor. The backends all run in user mode. There is also some kind of support for profiling and debugging. Basically, RapidMind targets the lowest hanging parallel fruits (see §5.7). RapidMind's main competitor PeakStream was bought by Google in 2007 and to my knowledge there are no similar libraries on the market

6.3.3 CUDA, Brook+, Intel Ct

CUDA and Brook+ are nVidia's and AMD's general purpose GPU (a.k.a. GPGPU) programming languages. They are both extensions of C which target data stream programming, that is DLP. Intel is developing a language called Ct, which is also targeting streaming (Ct is short for C for Throughput), but for CPUs. Just like Brook+ and CUDA, Ct is an extension to C but also to C++. No specifications of the language have been released as of this writing. It is unclear how Intel Ct will relate to TBB and the existing desktop software stack (see §5.3.6) but it seems likely that it will be targeting Larrabee (see §6.4.2).

A library Supports Intel/AMD CPUs, nVidia and ATI/AMD GPUs and the cell processor.

6.3.4 Google MapReduce

Google has released a paper on its MapReduce framework, which targets huge amounts of data. Here is an excerpt from the abstract:

> MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

> Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

The same programming model can also be used for high performance NUMA machines as well as ordinary desktop shared memory machines, although the implementation will differ. The full text version of the paper is available at http://labs.google.com/papers/mapreduce.html. The MapReduce implementation described in the paper is a good example of a parallel program, as it tackles many common parallel problems.

6.3.5 Erlang

Erlang is to my knowledge the most widespread general purpose language which is not based on shared memory. It is a functional language based on asynchronous point-to-point and broadcast message passing[1] between really lightweight processes. Erlang was originally developed for the telecom industry and is therefore highly fault tolerant and robust. Processes, which can number in millions, monitor each other to keep track and clean up those who have crashed.

Processes can share collections ordered and unordered sets and multisets called Erlang Term Storage (ETS). They are key-value blackboards which can be opened in process-private, single-writer-multiple-reader or even multi-writer modes. Processes have to manage synchronization themselves if an ETS is being written to while others are accessing at. ETS implemented on top of shared memory can be efficient for CMPs but they can also be implemented in distributed setups.

Erlang is available for mainstream desktop OSes and can make use of SMP multicore processors. Needless to say, Erlang processes are not modeled by threads or processes on desktop OSes. The desktop community does not seem to be very wide, for instance a good GUI library seems to be lacking but you can use Java GUIs through JInterface or use a web server and a web-based GUI.

Erlang focuses on concurrency rather than parallelism (see §2.1). It seems well suited for task level parallelism and synchronization intense applications but to my knowledge there is no concept of data streaming or other DLP constructs. Erlang has a database though, called Mnesia, which can be stored in memory.

6.3.6 Singularity

Singularity is an operating system and a microkernel being developed at Microsoft Research which is focused on reliability. Lots of really lightweight so

[1] As opposed to synchronous message passing. See section §6.5.5) for an introduction.

called Software-Isolated Processes (SIPs) are statically checked at load time to not contain any security risks. SIPs may only use message passing to communicate with each other through protocols called contracts. The contracts are similar to *design by contract* but has also got an explicit notion of state used to determine when messages may and may not be sent. It is statically checked that SIPs can never share memory, but they can allocate and manipulate large data chunks which are then *moved* (see §5.4.4) to another SIP via a message.

Singularity faces a complete rewrite of the entire software stack, something many prominent people have been advocating for a long time. With today's virtualization technology we can hopefully find a way to execute contemporary software efficiently in parallel with a new software stack (see §5.3.6). Singularity is not about parallelism at all (see §2.1), but like many other approaches to reliable programming, lots of extractable parallelism falls out for free.

http://www.research.microsoft.com/os/singularity/publications/OSR2007_RethinkingSoftwareStack.pdf

6.4 Contemporary hardware projects

Some heterogeneous processors and some domain specific manycore processors are already on the market. This section contains is a small list of notable hardware projects targeting parallelism. Some have already shipped and some will ship soon.

6.4.1 Intel Atom

A new x86-processor for embedded computers focused on low power and low production cost that will run at 0.8 to 1.87 GHz. It is the first in-order (see §1.1.2) x86 processor in 15 years. There will be a single core and a dual-core version specified to consume a 2.5 and 4 watts respectively.

6.4.2 Intel Larrabee

Intel has announced a manycore GPU codenamed Larrabee rumored to have up to 48 identical cores with cache coherent *shared memory* supporting the x86 instruction set. Figure 1 in [25] features a simple 2D-schematic how cores, caches and interconnects will be laid out. nVidia was not late to critize the design [18].

6.4.3 IBM Cell

The Cell processor has one general purpose double-threaded core controls eight cores specialized in floating point calculations. It is used in PlayStation 3 and RoadRunner, the world's to date fastest super computer. Cell has said to prioritize bandwidth over latency and peak computational throughput over simplicity of program code. A schematic of cores, caches and I/O can be found on page 3 of [24].

6.4.4 AMD Torrenza

Torrenza is an upcoming hardware platform based on HyperTransport, a high-bandwidth, low-latency point-to-point interconnect. The idea is that third parties shall develop accelerators, such as GPUS and physics processors, and connect them to AMDs x86 Opteron CPUs.

6.4.5 AMD Advanced Synchronization Facility

AMD has been researching an extension to the x86 instruction set called Advanced Synchronization Facility (ASF). It provides a new kind of explicit hardware synchronization and aims to simplify and speed up lock-free programming and transactional memory implementations.

> ASF works by allowing software to declare critical sections that modify a specified set of protected memory locations. Protected memory that critical sections modify will become visible to other CPUs either all at once (when the critical section finishes successfully) or never (if the critical section is aborted). CPUs can protect and speculatively modify up to 8 memory objects that can each be at most cache-line sized and need to be size-aligned. When ASF detects conflicting accesses to one of these objects, it aborts the critical section. Unlike traditional critical sections, ASF critical sections do not require mutual exclusion. Multiple ASF critical sections on different CPUs can be active at the same time, allowing greater parallelism.

The full paper is available at
http://www.amd64.org/fileadmin/user_upload/pub/epham08-asf-eval.pdf

6.4.6 Sun Rock

Rock is a 16-core processor laid out in clusters of four cores. It is the first processor that will have hardware support for transactional memory (see §4.5). Additionally, it supports thread level speculative execution (see §5.5.2) and so called *hardware scouting*. In hardware scouting, a thread that is stalled by cache misses continues to execute in a side-effect free *run-ahead* mode until the memory of the cache miss becomes available. The idea is to increase the MLP (see §2.1.2) by running in to more cache misses that are fetched immediately and does not cause real or as long cache misses once the actual execution is resumed. Sun expects to ship Rock in 2009 according to The Register.

A 50 min video presentation of Rock's transactional memory is available at the link below. It reveals some interesting details and some problems the engineers ran into while integrating and testing a software emulation of Rock in Java. For instance, it seems like Rock will not support nested transactions.

http://blogs.sun.com/HPC/entry/video_transactional_memory_on_rock

6.4.7 Azul Systems

As a mention of domain specific data parallelism, Azul Systems offers 54-core processors with up to 16 processors per machine aimed at the server market.

6.4.8 RAMP

Developing a real-world manycore processor costs enormous amounts of money, money researchers do not have. To allow researchers to explore the design space of manycore processors and software stacks built on top of them, the Research Accelerator for Multiple Processors (RAMP) project was formed. RAMP is an umbrella for multiple projects which emulate manycore processors with FPGAs. An FPGA is in essence a cheap and reprogrammable chip, powerful enough to build simpler processors.

6.5 Pointers to more information

This section contains a small selection of interesting popular science material relating to the manycore shift. Many of them are informal interviews with some of the industry's leading parallelization experts. All of them are quite easy to digest, only one research article is listed.

6.5.1 Intel Go Parallel, Are multicore processors here to stay?

Intel has a series videos called Go Parallel. "'Are multicore processors here to stay?"' is one of them. It explains why processors need to turn to parallelism to deliver notable performance increases in about 8 minutes.

http://www.devx.com/go-parallel/Door/36013

6.5.2 Channel 9 interview with Burton Smith

A one hour long informal video interview with a High Performance Computing veteran on past HPC parallelism and future desktop parallelism.

http://channel9.msdn.com/shows/Going+Deep/Burton-Smith-On-General-Purpose-Super-Computing-and-the-History-and-Future-of-Parallelism/

6.5.3 Channel 9 interview with Dan Reed

A half an hour video interview with the director of Microsoft's scalable / multicore systems research focused on server-side parallelism.

http://channel9.msdn.com/shows/Going+Deep/Dan-Reed-On-the-ManyCore-Future-and-Parallelism-in-the-Sky/

6.5.4 Google Tech Talk, Getting C++ threads right

One of the driving forces behind C++'s new memory model explains why we should bother with threads, some thread related bugs they have found in important standards and some of the issues with adding a memory model to a language like C++. About one hour long video presentation.

http://www.youtube.com/watch?v=mrvAqvtWYb4

6.5.5 Google Tech Talk, Advanced topics in programming languages: Concurrency/message passing

An introduction to data flow/message passing which exemplifies how programming models could look like in contrast to today's shared memory and imperative languages. One hour long video presentation with Rob Pike, co-inventor of UTF-8 and engineer on Bell Labs' famous Plan 9 operating system which aimed to replace UNIX.

http://video.google.com/videoplay?docid=810232012617965344

6.5.6 Simon Peyton Jones on transactional memory

15 minute video presentation which explains what transactional memory is and what problems it can solve.

http://blip.tv/file/317758/

6.5.7 A view from Berkeley

A multidisciplinary group of researchers spent two years discussing the shift toward parallelism and this paper summarizes their findings.

http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

6.5.8 Interview with Jay Hoeflinger automatic parallelization

An interview which explains the major problems of automatically extracting parallelism at compile time.

http://www.thinkingparallel.com/2007/08/14/
an-interview-with-dr-jay-hoeflinger-about-automatic-parallelization/

6.5.9 Embedded computing webinar on multicore shift

A group of parallel experts discuss their view on the most important parallelization problems we are facing in an one hour long phone webinar. Embedded computing it is not discussed much and virtually everything is applicable to desktop parallelism. Requires TechOnline registration.

http://www.techonline.com/learning/webinar/208400266

6.5.10 How to split a problem into tasks

A brief guide on how to split work into parallelizable tasks.
http://www.thinkingparallel.com/2007/09/06/how-to-split-a-problem-into-tasks/

6.5.11 The Next Mainstream Programming Languages: A Game Developer's Perspective

Tim Sweeney, founder of Epic Games and the Unreal Engine, talks about parallelism in today's games and how he would like languages to evolve to make game development more productive, reliable and adapted to parallelism. 67 page self explanatory powerpoint slide.
http://www.cs.princeton.edu/ dpw/popl/06/Tim-POPL.ppt

Part II

Zooming in on C++0x's memory model

Chapter 7

Introduction

The second part of this thesis zooms in on a central topic in shared memory programming models; memory models. More specifically, it explains how the upcoming C++ memory model works. This memory model is interesting since it combines many memory model approaches. If you understand it, you understand a lot about memory models.

The C++ memory model and terminology is related to some widespread consistency models used in the research community. The ambition is to bridge the gap in terminology between the research community and the C++ community. Finally, the memory model is applied to lock-free programming and shows what a hazard pointer implementation might look like.

Most programmers will not need to bother with memory models, they can rely on higher level abstractions and libraries. This part is intended for library developers, researchers and people who like to go deep[1].

7.1 Data consistency models

Researchers believed that uniprocessors would *soon* need to become parallel for a long period of time. Few people believed they would remain dominant and increase performance for as many years as they did. *Data consistency models* have been researched for decades. They basically state how different processors are allowed to have different views of the state of a program. Since desktop processors have been single core for a long time, data consistency software models have primarily been applied to high performance computing (HPC) and more exotic platforms. Now that we hit the three walls (see §1.1, part 1) and have to turn to parallelism, this research has become highly relevant.

The data consistency models were originally developed for distributed computing and message passing systems. But these models can be applied to shared memory chip level multiprocessors (CMPs) too. Processors are replaced by

[1] Read nerds

threads or cores. Message sending and receiving are replaced by writing and reading shared memory.

7.2 Introducing C++0x

The current C++ standard is entirely single-threaded. Existing multithreaded programs and libraries are based on compiler specific extensions. With the ongoing shift toward parallelism, there is a huge need for a portable and well designed multithreading version of C++. The most difficult part of such a specification is the memory model (see §4.2.4, part 1) as C++ aims to be efficient on almost all hardware architectures. Three independent groups, the C++ standards committee, Microsoft and the OpenMP (see §6.2.4, part 1) group, realized this need and started working on a memory model for C++ [41].

The upcoming C++ standard, called C++0x, introduces the notion of threads and includes a memory model. The standard was planned to be released at latest 2009 (hence the 0x), but might not be released in time. It is still being worked on but is fairly stable. Therefore, some of the covered memory model details might change.

Chapter 8

Classical consistency models

What happens when several threads write and read from the same memory location? If there is no caching or redundant copies of the same data it is easy. The content of the single memory location will always represent the data. What happens when there are several copies of the same data stored in different memory locations? Well, then this data must be kept synchronized. The problem is that this synchronization can be very expensive. For instance, updating a super fast small cache memory from a much slower, but larger, main memory is naturally expensive.

Another problem is that the compiler and processor want to reorder instructions to optimize the code and increase instruction level parallelism (ILP). Reorderings which are totally safe for single-threaded programs might not be ok for multithreaded ones. How shall the compiler and processor know which instructions it can reorder?

On the one hand, full synchronization and no reorderings make multithreaded programs easy to write and understand. On the other hand, excessive synchronizations, lost optimizations and lost ILP makes programs much slower. There is a trade-off between ease of programming and efficiency. This is where data consistency models are useful. A data consistency model states which reorderings are allowed, which writes must be observed by which loads and when synchronizations will occur. This section explains the classical data consistency models called sequential consistency and linearizability. Data consistency models can also be used for reasoning about distributed systems. When they are applied to shared memory systems they are called memory models.

8.1 Shared objects

Objects which are accessed by multiple threads are called *shared objects*. A *sequential object* designed for single-threaded use can safely assume that only

one method call is accessing the object at any given time. A shared object must cope with multiple threads simultaneously calling its methods. Consider a concurrent stack. If two threads are in the middle of overlapping push calls, which element will be pushed first? What is the size of the stack from the point of a third thread? If there are two stack objects, will other threads see modifications to these stacks in the same order? To not lose our minds, we would like shared objects' method calls to appear to happen in a one-at-a-time, sequential order.

8.2 Program order

The order in which imperative program code is written is called *program order*.

8.3 Weak consistency

Definition Shared objects whose method calls:

- Appear to happen in a one-at-a-time sequential order
- Are seen in the same order by all threads

are said to provide *weak consistency* [30]

Weak consistency ensures that no temporary states, which might violate invariants, are ever seen. It also ensures that the order method calls take effect in, is shared between threads. However, this order is undefined and can be entirely unrelated to program order. This makes weak consistency rather useless in itself. Instead, it is used as a building block of more useful consistency models.

8.4 Quiescent consistency

A shared object is said to be *quiescent* if none of its methods are being called at some point in time.

Definition Shared objects which promise:

- Weak consistency
- That method calls separated by a period of quiescence should appear to take place in their real-time order

are said to provide *quiescent consistency*.

Quiescent consistency can make sense for objects such as shared counters. A number of threads count occurrences of something. While the counter is being incremented, there might not be an interest in its value. After each thread has stopped counting it notifies a logger which reads the value after the last thread has stopped counting.

8.5 Sequential consistency

Definition Shared objects which promise:

- Weak consistency
- That method calls should appear to take place in program order

are said to provide *sequential consistency*.

Sequential consistency is a very intuitive consistency model for a programmer - whatever command issued first should take effect first.

8.6 Linearizability

Definition Shared objects which promise:

- Weak consistency
- That method calls should appear to take place in some *instant* between method invocation and response

are said to be *linearizable*.

This imposes a real-time order between method calls. The precise definition of this order is called the *precedence relation* and can be found in [31]. Linearizable shared objects provide sequential consistency since real-time order implies program-order, assuming all linearizable method calls are executed in program order.

8.7 Composability

So far we have been talking about what reordering guarantees a single shared object provides with respect to its method calls. What happens if we have several shared objects involved? First of all, let us note that we can talk about the same consistency guarantees for an execution of multiple shared objects or a single object.

If we compose an execution of two shared objects which both provide weak consistency, will the composed system also provide weak consistency? The answer is yes, having multiple objects will not break the promise that each method call will appear to take place in some sequential order for each shared object. All of these orders are shared between all the threads, so that promise is not broken either.

What about quiescent consistency? This is also composable. The system is not in a quiescent state when any of the shared objects is being called. In this case we need not give any ordering guarantees. When the system is in a quiescent state all shared objects are too. At this point all method calls on all

variable will appear to have taken effect in some undefined order. Hence, the system as a whole provides quiescent consistency.

And sequential consistency? Unfortunately this consistency model is not composable. Consider the following example:

Thread 1 executes	Thread 2 executes
`int r1 = y.get();`	`int r2 = x.get();`
`x.set(r1);`	`y.set(42);`

Table 8.1: Code which can lead to unexpected results

If x and y are two sequentially consistent integer objects which initially are zero, what possible values can r1 and r2 assume? Quite unintuitively an execution can produce the final result r1 = r2 = 42. Sequential consistency shared objects promise that program order is preserved for each object *individually*. Since neither of the threads access the same shared integer twice, all method calls can be reordered arbitrarily. The method calls can take effect in the following order:

```
1  y.set(42);
2  int r1 = y.get();
3  x.set(r1);
4  int r2 = x.get();
```

Figure 8.1: Allowed order in which method calls may take effect if x and y are sequentially consistent objects

This reordering violates the total program order for thread 2 and is thus not a sequentially consistent execution.

Linearizability, on the other hand, is composable. It imposes a global reordering constraint - the real-time order - on all method calls, regardless of which shared object they belong to. No matter how many shared objects there is and no matter how many methods are called, all threads will see all method calls taking effect in the same order.

8.8 Implications of composability

We ideally want method calls to appear to take effect in an order consistent with the program orders of all threads. In other words we want all executions to be sequentially consistent, regardless how many shared objects are involved. This corresponds well to the human intuition that each thread takes one step at a time. Linearizability provides this but can be expensive or even impossible to implement given a particular CMP (see §12.6). Sequentially consistent objects do not provide this. They must in some way be implemented together to provide

composed sequential consistency[1]. This makes it more difficult to implement and proof the correctness as a program, as you can not consider individual objects in isolation [2].

In between linearizable objects and sequentially consistent objects, there is a wide range of other consistency models which only prevent certain method call reorderings. They provide a trade-off between the cost of synchronization and ease of programming.

[1] C++0x uses an internally hidden global order, shared between all SC operations, to achieve this. See 11.5.

Chapter 9

C++0x memory model and atomic API

The upcoming version of C++, called C++0x, introduces the notion of threads and an accompanying memory model. It includes a rather rich atomic interface and multiple consistency models to choose from. This section will introduce the memory model and parts of the atomic API. The next chapter will relate them to the classical definitions introduced in the previous chapter.

9.1 Goals

The C++ standard committee has a wide range of goals for C++0x's threading ambition [8]. Knowing these can help understand some of the design choices:

- Provide the primitives critical to inter-thread memory synchronization.
- Provide the primitives critical to effective lock-free algorithms.
- Provide a very high degree of C and C++ interoperability.
- Cover the common range of existing hardware primitives.
- Expose those hardware primitives efficiently to programmers, that is, the design should minimize overhead.
- Provide a syntax that reduces errors.
- Support generic programming.

9.2 Wording

Before continuing, a word of caution is in place. The words used in this chapter are those used by the C++ standards committee. These are not necessarily in

sync with the wording of the research community. For instance, C++0x's atomic objects using relaxed memory order closely resembles sequentially consistent shared objects (see §11.4) while C++0x's sequential consistency memory order is similar to linearizability (see §11.5). The N2691 draft C++ standard seems to define what it means that two operations "'precede"' each other in a very different way compared to Lamport's temporal precedence definition (cf. 1.9.p14 N2691 and [31]). Other concepts might have more subtle differences in wording too.

Chapter 10 explains why you cannot use some common methods when analyzing C++0x programs. C++0x's abstractions are compared and related to the classical consistency models in chapter 11.

9.3 Atomic types and operations

C++0x provides a set of *atomic types* such as atomic_flag, atomic_bool, atomic_int and atomic_address. An instance of an atomic type is called an *atomic object*. Each atomic object represents a single *memory location*. The atomic objects provide some *atomic operations* on its memory location. Atomic operations provide mutual exclusion to the memory location, typically by mapping to a single atomic hardware instruction.

There is a also an atomic<T> template for certain user defined types which provides a limited set of operations. Which operations are supplied depend on the atomic type but all atomic operations are of one of these three types:

- Store
- Load
- Read modify write (RMW)

Store operations perform a write to the memory location and load reads the memory. The available RMW operations are swap, test_and_set compare_and_swap, fetch_add, fetch_sub, fetch_and, fetch_or and fetch_xor.

Store operations and RMW operations which store a new value are called *writes*. Load operations and RMW operations are called *reads*.

9.3.1 Memory orders and fences

C++0x's *memory orders* are similar to traditional consistency models, as we will see in the next chapter. They decide which reorderings of operations are allowed, which is then used to analyze allowed program behavior. You select which memory order you want *per atomic operation* rather than per atomic

object. You can choose between:

- Relaxed memory order
- Consume memory order
- Acquire memory order
- Release memory order
- Acquire+release memory order
- Sequentially consistent memory order

The different memory orders are explained in detail later. A quick outline is given here; Relaxed memory order does not introduce any synchronization between threads and allows a lot of reorderings. Consume memory order is beyond the scope of this paper but is briefly explained in section 9.10. Acquire and release memory order synchronize writes and loads on a certain memory location in a pair-wise message-passing fashion. Sequentially consistent (SC) memory order introduces a global order between all SC operations even when they operate on different atomic objects.

The atomic API also offers two traditional global fence (a.k.a. memory barrier) operations, atomic_memory_fence and atomic_compiler_fence. These fences prevent reorderings in the thread that execute them. They provide a more explicit alternative than memory order-tagged operations and can be used when porting multithreaded fence-based programs to C++0x.

9.4 Happens-before relation

The most central part of C++0x's memory model is the *happens-before relation* ($\rightarrow_{hb}$). It states in which order different operations should appear to happen with respect to each other for a given execution.

Definition Two operations, A and B, are *happens-before related* if either of the following holds:

- A is sequenced before B in program code, or (Program order)
- A *synchronizes-with* B, or (explained later in §9.6)
- A $\rightarrow_{hb}$ X and X $\rightarrow_{hb}$ B (Transitivity)

Happens-before is an *ordering*, meaning no circular relations are allowed.

Just like in Java [26], the happens-before order is the transitive closure of program order and the synchronizes-with relation.

Note that which happens-before relation exists depend on *runtime* conditions. When a function conditionally calls a subfunction, the executed operations of both functions become happens-before related. When a thread reads

a value that another thread has written, the executed atomic operations of the two threads can[1] become happens-before related.

Given two operations, A $\rightarrow_{hb}$ B, if B does not depend on a side-effect of A, B can be executed before A. The exact definition of side-effect is implementation dependent, but the idea is clear: compilers (at compile-time) and hardware (at run-time) may not reorder, interleave or otherwise perform magic on any operations so that they violate happens-before in an observable way. Consider the following single-threaded example:

```
1  int x=0, y=0, z=0;
2  x=1;
3  y=2;
4  z=x+y;
```

Line 2 and 3 may be reordered even though 2 $\rightarrow_{hb}$ 3 by program order. This is because line 3 does not depend on line 2, to line 3 it appears as all previous lines have been executed. On the other hand, line 4 can not be reordered since line 2 and 3 happens-before it and line 4 depends on both. By the same reasoning, line 2 and 3 can safely be executed in parallel or out-of-order in a processors pipeline.

9.5 Data race

In general, a data race occurs when one thread writes a variable and another read or write it without proper synchronization. The exact definition of a C++0x data race [8] is:

Definition An execution of a program contains a data race when:

- Two threads access the same memory location
- At least one of the accesses is not an atomic operation
- At least one of the accesses is a write
- There is no happens-before relation between the accesses

Data races are also runtime creatures. Rather than saying a program contains a data race you say that it contains a risk of a data race. Data races in C++0x have undefined behavior. Basically, any two operations that risk becoming a data race should be considered a serious bug. Data races can be avoided by using atomic operations to introduce synchronizes-with edges between the accessing threads (see §13.3.4 for an example). Traditional locks work because they use atomic operations, which do exactly this, internally.

Java has to allow data races for security reasons while C++ consider them to be bugs. The full rationale why data races are left undefined can be found at [5].

[1] For instance, if the write operation has release semantics and the load operations has acquire semantics.

9.6 Synchronizes-with relation

The synchronizes-with relation introduces a happens-before relation between two atomic operations executed by different threads. When a store with memory order release from one thread is observed by a load with memory order acquire from another thread the store happens-before the load.

All synchronization between threads comes from the synchronizes-with relation. Locking/unlocking mutexes, thread launching/joining and other library features which are used for concurrent synchronization are implemented in terms of atomic operations.

9.6.1 Synchronizes-with wording

A SC or release memory order store is said to have *release semantics*. A SC or acquire memory order load is said to have *acquire semantics*. RMW operations with SC or acquire+release memory order have both acquire and release semantics. An operation with release semantics is called a *release operation* and acquire semantics operations are called *acquire operations*.

A release store synchronizes-with all acquire loads that observe the written value. You do not say that loads synchronize with stores, it is always stores that synchronize with loads.

The precise definition of synchronizes-with is a little bit more complicated. A load which observes a store and the modification of a later RMW operation should happen after both operations. Both the store and the RMW operation should synchronize-with the load. To encompass this case, the definition of synchronizes-with introduces the notion of a *release sequence*.

Before giving the definition of a release sequence and synchronizes-with we need to understand another important concept called *modification order*.

9.7 Modification order

During an execution, each atomic object has its own *modification order*. This is a list of all the modifications to its memory location which occurred during the execution. The modification order is like a time-line for an atomic object:

- The modification order is the same for all threads
- If a thread has seen a certain write then it cannot subsequently see another write which occurred earlier in the modification order

The modification order rules always apply, even if you use memory_order_relaxed.

The modification order rules are rather weak by themselves. Thread 1 might see the latest modification of an atomic object x and an old modification of object y at the same time as thread 2 sees an old modification of x and the latest of y. Each thread has its own *view* of a particular modification order. So how do we synchronize the views of different threads? Apart from the modification

order rules, all stores and loads obey the happens-before ordering rules which can relate different threads. For instance, if a value A is loaded from an atomic object and the load happens-before another load which sees a value B, then B must be the same value as A or one which occurs later in the modification order.

When reading a non-atomic memory location, the latest preceding write according to happens-before must be returned.

9.8 Release sequence and synchronizes-with definition

The release sequence is a part of the synchronizes-with relation which in turn is a part of the happens-before relation. Its only importance is to help explain exactly which operations synchronize with each other. Understanding these definition can be useful to avoid unnecessary happens-before synchronization, such as between two consumers or between two producers.

Definition A release operation A *synchronizes-with* an acquire operation B that reads a value written by any side effect in the release sequence headed by A.

Definition A *release sequence* on an atomic object M is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first operation is a release, and every subsequent operation is:

- Performed by the same thread that performed the release, or
- A non-relaxed read-modify-write operation.

These definitions ensure that both the latest store and subsequent RMW operations synchronize with a load that sees the effect of all of them. For instance, consider a modification order where thread 1 release-stores 10, then thread 2 release+acquire increments the value by one and finally a third thread acquire-loads 11. Both the first and second thread will then synchronize with the third thread.

To understand the definitions better, let us have a look at another longer example.

9.8.1 Release sequence example

Assume we have an atomic_int with the modification order described in table 9.1.

This means the atomic_int had the values 10, 20, 30, 40, 50 and 60 during this particular execution. What are the release sequences headed by the different lines? The release sequence headed by the modification #1 is just #1 since the next modification is a release performed by another thread. For #2, the release sequence is the modifications at line #2-#5. Note that no-one can synchronize with #3 since this is an acquire operation. Therefore no release sequence is

#	modification	memory order
1	Thread 1 stores 10	memory_order_release
2	Thread 2 stores 20	memory_order_release
3	Thread 3 fetch_and_adds 10	memory_order_acquire
4	Thread 2 stores 40	memory_order_release
5	Thread 1 fetch_and_adds 10	memory_order_acq_rel
6	Thread 1 stores 60	memory_order_release

Table 9.1: Example modification order of an atomic_int

headed by #3. The release sequence headed by #4 is #4, #5 since #5 is an RMW operation with non-relaxed memory order. #5's headed release sequence includes #6 since #6 is a release operation from the same thread.

Table 9.2 summarizes all release sequences headed by each modification in Table 9.1. It also states which stores synchronize with acquire loads that observe a particular value.

#	headed release sequence	synchronizes-with	acquire load that observe
1	#1	#1	10
2	#2,#3,#4,#5	#2	20
3	None	#2	30
4	#4,#5	#2,#4	40
5	#5,#6	#2,#4,#5	50
6	#6	#5,#6	60

Table 9.2: Resulting release sequences and synchronizations of table 9.1

For instance, if an acquire load observes the value 40, the store #2 and store #4 will synchronize with the load. This is rather intuitive since modification #2 was store of 20 which was later incremented by modification #4. It was also incremented by modification #3 but since this is not a release operation, it does not synchronize with any loads.

Also note that, modification #3 and #5, themselves are acquire operations. Therefore #2, which observed 20, synchronizes-with #3. #5 observed 40 before applying its modification and therefore synchronizes with both modifications #2 and #4.

9.9 Sequentially consistent ordering

For completeness, the precise semantics of C++0x SC operations [3] is supplied:

Definition There is a single total order S on all memory_order_seq_cst operations, consistent with the happens-before order and modification orders for all

affected locations, such that each memory_order_seq_cst operation observes either the last preceding modification according to this order S, or the result of an operation that is not memory_order_seq_cst

SC stores have release semantics, SC loads have acquire semantics and SC RMW operations have both acquire and release semantics. In practice, two memory_order_seq_cst operations will never be reordered so that program behavior is affected.

9.10 Consume memory order

As of this writing, the C++ committee has just voted to add an additional memory order called consume memory order. The rationale behind this is explained in technical report N2664 [42]:

> There are two significant use cases where the current working draft (N2461) does not support scalability near that possible on some existing hardware.
>
> **read access to rarely written concurrent data structures** Rarely written concurrent data structures are quite common, both in operating-system kernels and in server-style applications. Examples include data structures representing outside state (such as routing tables), software configuration (modules currently loaded), hardware configuration (storage device currently in use), and security policies (access control permissions, firewall rules). Read-to-write ratios well in excess of a billion to one are quite common.
>
> **publish-subscribe semantics for pointer-mediated publication** Much communication between threads is pointer-mediated, in which the producer publishes a pointer through which the consumer can access information. Access to that data is possible without full acquire semantics.
>
> In such cases, use of inter-thread data-dependency ordering has resulted in order-of-magnitude speedups and similar improvements in scalability on machines that support inter-thread data-dependency ordering. Such speedups are possible because such machines can avoid the expensive lock acquisitions, atomic instructions, or memory fences that are otherwise required.

The consume memory order introduces as much as three new relations which describes how data dependencies propagate through a program. Since the definitions are rather complicated and might very well be subject to change, they are not included in this thesis.

9.11 Summary of memory model

Excluding fences and consume_memory_order, there are three central orderings which determine the allowed execution behavior; The happens-before relation which affect all operations, the global order of all SC atomic operations and the modification order of each atomic object. The SC order and the modification orders and its associated views are quite easy to understand and reason about. The program order part of happens-before is also easy to grasp. But through multiple synchronizations and transitivity, operations can become happens before-related in quite tangled and complex ways. Such cases often require careful thought and analysis.

Which ordering guarantees you get depends on which memory order you select for each atomic operation.

9.12 The actual standard

The ordering constraints given in this chapter are the most important ones. They are somewhat simplified rules based on primarily N2691, an early draft of the C++ standard. Some of the actual definitions in the draft are rather gory and build on a lot of implicit assumptions which might not be obvious to new readers. Some of the definitions are defined in terms of other abstractions which are not properly defined at all (such as "'value computation"'). Other definitions are circular (cf. notes of 1.10.p13). One definition allows a very strange execution but is then patched by a note stating that this particular example should not be allowed (cf. 29.1.p5). Hopefully, the definition screws will be tightened before the actual standard is released. For now, the standard is padded with lots of notes which help explain the intended semantics of the definitions.

Section 1.9, 1.10 and 29.1 of the N2691 standard are the central sections of the memory model.

Chapter 10

Analyzing C++0x programs

C++0x's memory model defines how shared memory modifications from one thread becomes visible to operations in other threads. This implicit approach gives compilers and hardware a lot of freedom to perform optimizations behind the scenes, but it also invalidates a lot of human intuition. This chapters explains how multithreaded C++0x code can and cannot be analyzed.

10.1 Do not rely on invocation-response histories

One way to analyze data consistency is to write invocation-response histories. A number of threads have a number of shared objects which they use to communicate. Invoking a method call on a shared object is seen as sending a message to the shared object and the response is also seen as a message. An execution history is written down as a list of method invocations and responses, and the set of possible execution histories can be used to reason about possible and impossible results.

Thread	Shared object	Message
1	the_queue	put(100)
1	the_queue	return void
2	the_queue	pop
1	the_queue	put(200)
2	the_queue	return 100
1	the_queue	return void

Table 10.1: Example execution history

C++0x allows the compiler and hardware to reorder the operations and instructions of individual threads, even for single threaded programs. Therefore it is not sufficient to reason about how different threads become interleaved, you must also consider possible reorderings. On top of this, there is some freedom when writes to the shared memory locations become visible. These factors makes invocation-response histories a poor tool for reasoning about multithreaded C++0x programs.

10.2 Do not draw timelines

Another common approach to analyze consistency models is to use a timeline. A timeline per thread is drawn which follow the program order of the threads. Since individual threads may have their operations reordered, program order does not follow temporal order and timelines do not make very much sense. In fact, there is no notion of time in C++0x's memory model and no requirement that any shared memory modifications shall be consistent with real time.

10.3 Evaluation sequences

A lot of examples from the C++ standards committee involve some code which is executed and some kind of imagined history where each thread takes one step at a time. These types of examples are also common in the remainder of this thesis.

```
void thread1() {
  int y1 = y.load();
  x.store(y1);
}
void thread2() {
  int x2 = x.load();
  y.store(10);
}

// Evaluation sequence for one imagined execution
  y.store(10);
  int y1 = y.load();
  x.store(y1);
  int x2 = x.load();
```

The idea is that each operation sees the effect of all operations above it in the evaluation sequence. The operations of each thread need not follow program order as long as happens-before is not violated in an observable way and all other memory model rules are obeyed. The operations are ordered according to when they take effect and become visible to *all* other threads, not according to how they are executed.

C++0x requires (cf. N2691 1.9.p5) that any single store to an atomic object shall have a sequence of evaluations which explains the stored value. When you have multiple memory locations and multiple threads, it is less clear that you

can analyze all executions in terms of evaluation sequences. Nevertheless, you can use evaluation sequences as a tool to reason about certain executions.

There is another notion of *possible execution sequences* (cf. N2691 1.9.p5) which existed before threading were introduced in C++0x. It is unclear if they are applicable to multithreaded execution.

Chapter 11

Relation to classical definitions

The memory orders and other ordering constraints are very similar to the classical consistency models. But there are a number of important differences. Understanding these differences is useful in understanding the memory model even if you are not familiar with the classical definitions

11.1 Atomic object - shared object

Atomic objects are similar to shared objects but there are differences. Atomic operations are low-level while the methods of shared object can be high-level. C++0x's memory model is defined in terms of atomic operations. Non-atomic operations need to be properly happens-before related by atomic operations. As long as they are, you can lump several operations together into a higher-level operation and reason about it.

Each shared object implementation provides its own consistency model. In C++0x, the consistency guarantees you get depend on the memory order you select for each operation. You do not select a memory order per atomic object.

11.2 Happens-before - happened-before

C++0x's definition is very similar to Lamport's happened-before relation [30] although his definition is defined in terms of message passing. In a shared memory programming model, such as C++0x, the primary means of communication between threads is shared memory. That a thread A writes a value to a memory location which a thread B then reads can be seen as passing a message. Simply put, if A synchronizes-with B, A sends a message to B via shared memory.

Do not confuse happens-before with Lamport's precedence relation [31] which is a temporal order stating how operations are *actually* executed. C++0x's

memory model has no notion of time.

11.3 Modification orders - weak consistency

The method calls of a weakly consistent shared object (see §8.3) appear to happen in a one-at-a-time sequential order which is seen in the same order by all threads. A modification order is a one-at-a-time sequential order in which modifications to a particular atomic object are seen. This order is shared between all threads.

In other words; to have one modification order per memory location, without any happens-before or SC ordering requirements, is identical to weak consistency. This would allow all modifying operations on an atomic object to be reordered freely, as long as their order is consistent across threads. This is not a very useful consistency model and C++ does not offer anything this weak. All executions must obey the visibility constraints of happens-before. But just like weak consistency, the modification order rules serve as a building block for more useful memory orders. All atomic operations, regardless of the selected memory order, must obey the modification order rules.

11.4 Relaxed memory order - sequential consistency

Sequential consistency (see §8.5) shared objects promise weak consistency and that method calls should appear to take place in program order. Assuming there are no synchronizes-with relations, the happens-before rules describes exactly how operations should appear to take place in program order.

Imagine every operation on all atomic objects is atomic memory_order_relaxed operations. This means that there are no synchronizes-with relations and no SC ordering requirements. All atomic objects then behave exactly as a sequentially consistent shared object. Note that we are talking about sequential consistency *per* atomic object. Combining multiple atomic objects with only relaxed memory order would not guarantee a combined evaluation sequence which is sequentially consistent. Recall the example in table 8.1 in §8.7.

Another similar example is:

Thread 1 executes	Thread 2 executes
`x.store(true, memory_order_relaxed);`	`y.load(memory_order_relaxed);`
`y.store(true, memory_order_relaxed);`	`x.load(memory_order_relaxed);`

Table 11.1: Relaxed memory order example

Human intuition tells us that the x-store should happen before the y-store. But the two loads can very well see y as true and x as false, which is motivated

by the following, allowed, evaluation sequence:

```
x.load(memory_order_relaxed);
x.store(true, memory_order_relaxed);
y.store(true, memory_order_relaxed);
y.load(memory_order_relaxed);
```

Listing 11.1: Allowed evaluation sequence of table 11.1

Since program order is only respected for each individual memory location and no thread accesses the same location twice, all evaluation sequences are allowed. Relaxed memory order allows a lot of unintuitive reorderings and should be used with extreme care. An atomic store which does not synchronize-with an atomic load have no other visibility constraint than that it should become visible within a "'reasonable amount of time"' (cf. N2691 29.1.p6).

11.5 Sequentially consistent memory order - linearizability

C++0x's SC memory order has a global order for all atomic operations, regardless how many memory locations are involved. The global order is not real-time but an internal order, hidden away in the implementation of C++0x compilers.

There is a similarity between C++0x SC operations and linearizability (see §8.6), in the sense that SC operations provide a global order for all SC atomic operations. Just like linearizability, this order is consistent with all program orders of the participating threads and all modification orders of the participating atomic objects. A main difference is that linearizability is defined using a notion of real time, while C++ SC is defined in terms of allowed reorderings. The former is consistent with real-time and composes better with I/O operations while latter provides more reordering freedom (see §12.5 and §12.6).

The *combined* evaluation sequence of all SC atomic operations on *all* atomic objects is guaranteed to be sequentially consistent. Hence the term "'sequentially consistent memory order"'. It does not refer to a sequential consistency property of individual atomic objects.

11.6 Acquire-release memory order - release consistency

Acquire-release memory order is arguably the trickiest part of the C++0x memory model. They allow one thread's operations up to a certain point to appear to have taken effect before other operations in another thread. This is a much lighter form of synchronization than SC operations. A single thread synchronizes with others pair-wise through a memory location, rather than all threads synchronizing with all others through some global order.

Release consistency was first introduced in [14]. The classical definition is beyond the scope of this thesis. How C++0x's acquire-release consistency works in practice is explored in greater depth in the next chapter.

Chapter 12

Understanding the memory model

How do we use the atomic API in practice? First of all, most programmers can rely on higher level abstractions and do not need to call the atomic API directly. If they still do and performance is not a problem, sequentially consistent operations should be preferred. They are easy to reason about. When performance really matters, you should try to use weaker memory orders such as acquire-release and relaxed operations. Relaxed operations are really tricky, so a good first step is often to use acquire-release operations.

To see if acquire-release operations suffice for a particular problem, you first need to identify the dangerous behaviors you want to avoid. You can then analyze how different happens-before relations are spread between threads and verify that none of the dangerous behavior are allowed. The following sections contain some examples which illustrate this.

12.1 Example: Synchronization through another memory location

The most basic example is how the program order of two threads becomes synchronized by the transitivity of happens-before relation.

To see how this happens, let us revisit the example in table 11.1. We want to prevent all executions where the x-load observes false and the y-load returns true. If the stores would have been release operations and the loads acquire operations, can we prevent all such execution sequences? Yes, if the y-load observes the value (true) written by the other thread the following happens-

before relations exist:

1. x-store $\rightarrow_{hb}$ y-store (program order, thread 1)
2. The acquire load of y observes the release store of y.
   Hence, y-store $\rightarrow_{hb}$ y-load (synchronization, thread 1 and 2).
3. y-load $\rightarrow_{hb}$ x-load (program order, thread 2)
4. By 1, 2 and 3, x-store $\rightarrow_{hb}$ x-load (2 x transitivity)

The x-store $\rightarrow_{hb}$ the x-load and the x-load is clearly dependent on the x-store. Thus, the x-store must observe the x-load in any execution where the y-load observes the value of the y-store. Therefore the x-load must return true if the y-load returns true. Had we used relaxed memory order, there would not have been any synchronization and the x-store would not be happens-before related to the x-load.

12.2 Example: Non-circularity of happens-before relation

Since the happens-before relation is an ordering, it cannot be circular. Let us look another example where this fact prohibits some unintuitive evaluation sequences. Assume we have two atomic_ints, x and y which are accessed by two different threads:

Thread 1 executes	Thread 2 executes
`int r1 = y.load(memory_order_acquire);` `x.store(r1, memory_order_release);`	`int r2 = x.load(memory_order_acquire);` `y.store(42, memory_order_release);`

Table 12.1: Acquire-release memory order example

The question is: "'Can r2 see 42?"'. This would correspond to the following evaluation sequence:

```
y.store(42, memory_order_release);
int r1 = y.load( memory_order_acquire);
x.store(y1, memory_order_release);
int r2 = x.load( memory_order_acquire);
```

This evaluation sequence is not sequentially consistent, so will acquire-release consistency help us prohibit this situation? Yes. For x to assume the value 42

r1 must observe 42 and happen after the y-store. If the x-load also observed the value 42 the following $\rightarrow_{hb}$ relations would then exist:

1. y-load $\rightarrow_{hb}$ x-store (program order, thread 1)
2. The acquire load of x observes the release store of x
   Hence, x-store $\rightarrow_{hb}$ x-load (synchronization)
3. x-load $\rightarrow_{hb}$ y-store (program order, thread 2)
4. The acquire load of y observes the release store of y
   Hence, y-store $\rightarrow_{hb}$ y-load (synchronization)
5. y-load $\rightarrow_{hb}$ y-load (transitivity of 1, 2, 3 and 4)

These relations form a circle which is not allowed. If r2 observes 42, r1 must have observed a write which $\rightarrow_{hb}$ the x-load and this is a contradiction. The atomic API implementations will prohibit these evaluation sequences for you if you use acquire-release operations.

12.3 Example: Double store-load

In some cases, acquire-release consistency allows unintuitive evaluation sequences. Consider Dekker's example involving two atomic_bools which are initially false:

Thread 1 executes	Thread 2 executes
`x.store(true, memory_order_release);`	`y.store(true, memory_order_release);`
`y.load(memory_order_acquire);`	`x.load(memory_order_acquire);`

Table 12.2: Acquire-release example which allows unintuitive reordering

Human intuition tells us that when the either of the two loads occur, the previous write should have taken effect. But both loads may observe false without breaking any rules. If they do, the only happens-before relations that exist are x-store $\rightarrow_{hb}$ y-load and y-store $\rightarrow_{hb}$ x-load. But the y-load does not depend on any side effect of the x-store and neither does the other two happens-before related operations. Hence, it is perfectly possible that neither of the stores become visible to the loads. memory_order_seq_cst operations would have forced a total ordering of all operations and disallowed both loads from occurring before both writes.

12.4 Example: Independent reads of independent writes

So if we consistently use acquire-release operations, we cannot always find a sequentially consistent combined evaluation sequence. Another interesting example where this is the case is called Independent Reads of Independent Writes (IRIW). It shows how two writes from different threads can be seen in different orders from two other threads.

```
atomic_bool x = false, y = false;

void thread1() {
  x.store(true, memory_order_release);
}

void thread2() {
  y.store(true, memory_order_release);
}

void thread3() {
  bool x3 = x.load(memory_order_acquire);
  bool y3 = y.load(memory_order_acquire);
}

void thread4() {
  bool y4 = y.load(memory_order_acquire);
  bool x4 = x.load(memory_order_acquire);
}
```

It is possible that thread 3 sees x as true and *later* y as false while thread 4 sees y as true and later x as false. From thread 3's perspective it seems like the x-write occurred before the y-write. Thread 4 sees the opposite. This is quite unintuitive since you would expect the writes to take place in some order. Here is an example evaluation sequence which would give us that result:

```
bool x4 = x.load(memory_order_acquire);
x.store(true, memory_order_release);
bool x3 = x.load(memory_order_acquire);
bool y3 = y.load(memory_order_acquire);
y.store(true, memory_order_release);
bool y4 = y.load(memory_order_acquire);
```

This clearly violates the program order of thread 4 and is thus not a sequentially consistent evaluation sequence. But it is a perfectly allowed behavior since the x4-load does not depend on any side effect of the y4-load.

If this behavior is unacceptable for some program, SC operations can be used for all the loads. This would force a total order in which all loads were evaluated, consistent with the program order of both thread 3 and 4. SC operations are said to supply the IRIW guarantee.

12.5 Composability issues

What if you want to compose new C++0x components with older C++ components implemented on top of compiler or hardware specifics. How could you achieve this?

The happens-before relations, SC orders and modification orders are not exposed in any way. C++0x seems like a big monolithic system. Naturally, compiler or hardware specifics are outside the scope of C++ standard. But composability with existing multithreaded C++ code is going to be very important to allow a smooth transition to C++0x. It seems likely that compiler vendors will provide some guarantees and information on the matter. For instance, how the C++0x orders relate to pre-0x implementation specific operations and fences. Also, all atomic operations are tagged as volatile which means atomic operations will be consistent with the volatile access order which certain pre-0x compilers use for multithreading.

The concept of OS processes not present in C++0x. This introduces composability problems when two C++0x processes need to communicate through shared memory. Each C++0x process has its own monolithic ordering system. To facilitate shared memory inter-process communication without introducing the concept of processes, the C++ committee expresses their intent on how atomic operations should be implemented in [20].

12.6 Why is linearizability not provided?

Many people think that global sequential consistency is indistinguishable from linearizability and confuse the two. Even if a processor promises global sequential consistency on all shared memory operations, regardless which process or programming model is altering it, linearizability is still stronger. Linearizable shared memory is composable with linearizable I/O orders. For instance, clock cycle-grained global time queries are useful for parallel algorithms, profiling and debugging. Such time queries provide a consistent view of the world with linearizable shared memory, while global sequentially consistent memory cannot rely on time consistency. Another benefit is that linearizable programs running on network processors or GPUs could communicate with linearizable programs running on the main CPU without worrying about reordering problems.

If linearizability is so great, why does C++ not supply a linearizable memory order? The answer is simple; it is not C++'s place to require such things. C++ aims to be compatible with all mainstream hardware architectures, some of which do not provide linearizable shared memory operations. From a hardware point of view, linearizable shared memory seems to be associated with a cost [2].

Note that a C++0x implementation targeting a linearizable hardware architecture can very well promise that SC atomic operations are linearizable. Code that relies on such compiler/hardware specific promises will not be portable though.

Chapter 13

Applying the memory model to hazard pointers

Lock-free programming relies heavily on the underlying atomic operations and memory model. As such, it serves as a good example of applying the C++0x memory model and atomic API. This chapter discusses how hazard pointers can be implemented in C++0x and which memory orders are required to maintain consistency. Two basic lock-free containers, based on well known lock-free algorithms, are also implemented to illustrate lock-free programming using C++0x.

13.1 Complexities of C++0x

Lock-free programming in C++0x is going to be very, very difficult. Lock-free programming in any language is very difficult (see §4.3, part 1). In C++0x you need to understand a quite complex memory model and then use it to perform a less than trivial analysis that your tricky lock-free code is correct. C++ supports exceptions so you need to consider exception safety. And C++ is not garbage collected so you must implement your own thread safe memory reclamation scheme (see §4.4, part 1). You might also need to implement your own lock-free memory allocator instead of using the default new allocator (see §4.3.5, part 1). The upside is that C++ code can be made very efficient and that C++0x code will probably be efficiently portable to many hardware and software platforms.

13.1.1 Comments

The example code contains lots and lots of comments. In general, comments can be a sign of poor design. While there has only been so much effort put into to the design of the example code, one can expect any lock-free code to contain a lot of comments. The reason is that programming languages are not adapted to lock-free programming. There are lots of implicit assumptions on

what other threads might and might not be up to. Many, seemingly independent methods need to be considered together to reason about the correctness of lock-free code. Such considerations cannot be expressed directly in the language, but rather should be commented in a very disciplined manner.

13.2 Atomic API implementation

There is no C++0x compiler as of this writing and therefore no atomic API and memory model implementation. In fact, the memory model and entire C++0x specification is still being worked on. To remedy this, a very small subset of the atomic API is implemented on top of Microsoft Visual C++ and Windows OS specifics. The correctness or efficiency of this naive implementation is not discussed as it is less interesting for the topic of this thesis. For completeness, the code can be found in appendix A.

It is also worth noting that the atomic operations of C++0x need not map to lock-free hardware instructions. The only atomic operation that is required to be lock-free is atomic_flag's test_and_set and clear operation. Other atomic types provide a runtime, per object query, called is_lock_free [20]. This allows C++0x compilers to target simple processors which do not have atomic instruction support. There will also be some compile time macros to determine if atomic integers and atomic addresses are always, sometimes or never lock-free. The example code in this chapter assumes that all atomic objects provide lock-free implementations.

13.2.1 Spurious failures and compare_exchange

std::atomic_compare_exchange, C++0x's CAS, will map against a load-linked and a store-conditional instruction on certain architectures (see §4.3.4, part 1). These instructions sometimes have *spurious failures*, that is a store-conditional is allowed to fail in rare cases although the loaded link has not changed. To accommodate such architectures efficiently, std::atomic_compare_exchange also allows spurious failures and user code has to be adapted to this fact.

To write correct code you have to be able to detect spurious failures. Many parallel algorithms also require that CAS implementations return the old value when the comparison fails. Rather than returning a std::pair or std::tuple[1], std::atomic_compare_exchange overwrites the second comparand with the atomic object's value and returns a bool stating if the CAS was successful. If it fails without changing the comparand, we have a spurious failure.

```
namespace std {
  bool atomic_compare_exchange(volatile atomic_int* atomic,
                               int* comparand, int new_value);
}

void foo() {
  atomic_int atomic_object(rand()%10);
```

[1] Tuples will be added in C++0x

```
  int comparand = 5;

  bool succeeded = std::atomic_compare_exchange(&atomic_object,
                                                &comparand, 20);
  // comparand now contains atomic_object's
  // value at the time the CAS was executed

  // If the CAS failed and the comparand has
  // not been updated, we have a spurious failure
  bool spurious_failure = !succeeded && comparand == 5;
  bool normal_failure = !succeeded && comparand != 5;
}
```

There is a suggestion [9] to have a weak and a strong version of atomic_compare_exchange which explicitly allows or does not allow spurious failures.

13.3 Hazard pointers example

To not lose the lock-free property of what is being implemented, it is not sufficient to have a lock-free allocator, lock-free atomic operations and a lock-free algorithm. A lock-free memory reclamation technique must also be used. Hazard pointers [34] is a lock-free reclamation scheme which introduces an interesting memory model analysis when implemented in C++0x. The hazard pointer implementation is the most complicated part of the example code. Before continuing, please read the brief introduction to memory reclamation and hazard pointers in §4.4 and §4.4.3 in part 1.

There is a US patent claim on hazard pointers. Some kind of license or permission might be required to use hazard pointers in the United States.

13.3.1 Ignored aspects

The goal of the hazard pointer implementation is to apply C++0x's memory model at a somewhat larger example. Therefore, some aspects which would be very important in a real world implementation have been overlooked. For starters, the default new and delete operators are used in all example code. A lock-free allocator should be used instead (see §4.3.5, part 1).

Memory or cache-line contention effects of the actual hazard pointers is not addressed either. It is likely that the hazard pointers themselves become a source of contention since they are frequently accessed. C++0x is meant to be portable and does not model the underlying memory architecture whatsoever. Therefore there is no way of specifying where memory is laid out or know how much padding is needed to avoid potential cache-line contention (see §2.4.4, part I).

C++0x introduces a new concept of *thread local storage*, which is basically data that exist in one instance per thread. Thread local storage can be accessed from other threads via references. It is probably safe to assume that the compiler will try to lay out thread local storage on memory locations that are easily accessible for the core that is executing the thread. Hopefully, it will also lay

out the memory so that storage from one thread is far apart from other threads' storage to avoid false sharing (see §2.4.4, part 1).

A real world C++0x hazard pointers implementation should use thread local storage for all hazard pointers and unclaimed nodes belonging to a particular thread. This example hazard pointer implementation does not emulate thread local storage nor does it consider false sharing issues. Rather, it uses a dynamic pool of hazard pointers (hazard_pointer_pool) and unclaimed data (retired_nodes_pool) which are shared between all threads. This would be a disaster in a real world implementation.

13.3.2 The basic idea

While one thread is dereferencing a shared pointer, another thread might delete it. This is the basic problem that memory reclamation (see §4.4, part 1) tries to solve. The hazard pointer solution is to flag all accesses of shared pointers by setting a hazard pointer before dereferencing. The reclaiming thread checks that the pointer it is about to reclaim does not have any hazard pointers pointing to it. A simplified use case with only two threads might look something like this:

```
101 node* ptr = shared_node.load();
102 while (true) {
103   thread1_hazard_ptr.store(ptr);
104   node* verify_ptr = shared_node.load();
105   if (ptr == verify_ptr) break;
106   else ptr = verify_ptr;
107 }
108
109 // Safe to dereference ptr here
110
111 thread1_hazard_ptr.store(0);
```

Listing 13.1: Code executed by dereferencing thread

```
201 node* ptr_to_reclaim = shared_node.load();
202 shared_node.store(some_other_ptr);
203 node* hazard_ptr = thread1_hazard_ptr.load();
204 if (ptr_to_reclaim != hazard_ptr) delete ptr_to_reclaim;
205 else reclaim_later(ptr_to_reclaim);
```

Listing 13.2: Code executed by reclaiming thread

The dereferencing thread needs to load the shared pointer it wants to access. It must then set the hazard pointer to let a potentially reclaiming thread know that it is about to dereference the pointer. This involves *two* atomic operations. Because these two operations are not performed atomically, the loaded pointer could have been erased before the hazard pointer was set. To ensure this is not the case, the dereferencing thread reads the shared pointer again and verifies it has not changed.

Just like in the stack example (§4.3.3, part 1) we have an ABA situation that does not cause any problems (see §4.3.4, part 1).

The reclaiming thread reads the shared pointer and then removes the pointer by replacing it with some other pointer. Next it reads the hazard pointer of the

dereferencing thread. A real implementation would read all hazard pointers of all other threads. If the dereferencing thread has not marked the pointer as hazardous we can be sure it can never access it again, because the pointer has been replaced with some_other_ptr. Therefore it is safe to delete. If it is marked as hazardous, it is scheduled to be deleted later.

13.3.3 Memory model analysis

The reasoning in the last section was not entirely satisfactory. It did not consider the memory model and possible dangerous reorderings. It is crucial that:

- The hazard pointer store (line 103) is really visible to a reclaiming thread (line 203) before the verifying shared node is loaded (line 104).
- The reclaiming thread's removal of the shared node (line 202) is visible to a dereferencing thread (line 104) before the hazard pointer is loaded (line 203).
- The hazard pointer clear (line 111) is NOT visible to a reclaiming thread (line 203) until dereferencing is complete.

Let us call these three properties *hp-set-safely*, *shared-node-removed-safely* and *hp-cleared-safely*. As long as these three properties hold, a deleted node will not be dereferenced.

What about other dangerous behavior?

How can we be certain that there are no other dangerous reorderings or writes that are not visible? The best you can do is try to formally express the problem you are trying to solve, then formalize your reasoning why your code is correct and finally mathematically prove that your reasoning is sound. This is known as *formal verification* and is beyond the scope of this thesis. The informal reasoning in the initial text of §13.3.3 is all that is provided.

Selecting memory order

Assuming we use the same memory order for all operations, which one must we use to guarantee hp-set-safely, shared-node-removed-safely and hp-cleared-safely?

Relaxed memory order is clearly not going to cut it. The hazard pointer stores might never[2] become visible to a reclaiming thread since the two threads never synchronize. Acquire-release memory order is not sufficiently strong either. An imagined execution sequence can violate hp-set-safely (move line 103 below line 104 in an imagined evaluation sequence). Line 103 happens-before line 104 but line 104 does not depend on any side effect of line 103 and may therefore be reordered below 103.

[2]It should only become visible in a reasonable amount of time (see §11.4)

If all atomic operations use SC memory order, all of the atomic operations will take effect in program order. Therefore hp-set-safely and shared-node-removed-safely will be guaranteed. What about hp-cleared-safely, can it still take effect too early? If the hazard pointer load observes the second store of 0 (line 111) all operations in the safe-to-reference part happens-before the delete. Since dereferencing clearly depends on a side effect of delete, the deletion cannot occur before dereferencing. Note that we do not rely on SC ordering in this case, it is sufficient that the hazard pointer load has acquire semantics and the clearing hazard pointer store has release semantics to guarantee hp-reset-safely.

Explicit fences could also have been used. Perhaps even consume_memory_order.

13.3.4 Using acquire-release consistency instead

So acquire-release consistency guarantees hp-cleared-safely. SC operations guarantees hp-set-safely and shared-node-removed-safely, but they can be really expensive on certain architectures [6]. As it turns out, we can get away with only acquire-release consistency at the price of some extra atomic operations. Basically, we have two pairs of atomic operations which operate on the same two memory locations. Neither pair depends on a side effect of the other pair but we still want to make sure that their combined execution obeys program order (hp-set-safely and shared-node-removed-safely). A common trick in these situations to add a third dummy memory location and perform an acquire+release RMW operation on it. One of the competing threads must execute its RMW operation last which will introduce a synchronizes-with relation between the two threads. Without further ado, here is the code:

```
301 node* ptr = shared_node.load(memory_order_acquire);
302 while (true) {
303   thread1_hazard_ptr.store(ptr, memory_order_release);
304   dummy_sync_variable.fetch_and_add(1, memory_order_acq_rel);
305   node* verify_ptr = shared_node.load(memory_order_acquire);
306   if (ptr == verify_ptr) break;
307   else ptr = verify_ptr;
308 }
309
310 // Safe to dereference ptr here
311
312 thread1_hazard_ptr.store(0, memory_order_release);
```

Listing 13.3: Code executed by dereferencing thread

```
401 node* ptr_to_reclaim = shared_node.load(memory_order_acquire);
402 shared_node.store(some_other_ptr, memory_order_release);
403 dummy_sync_variable.fetch_and_add(1, memory_order_acq_rel);
404 node* hazard_ptr = thread1_hazard_ptr.load(memory_order_acquire);
405 if (ptr_to_reclaim != hazard_ptr) delete ptr_to_reclaim;
406 else reclaim_later(ptr_to_reclaim);
```

Listing 13.4: Code executed by reclaiming thread

Assume the reclaiming executed the increment first. Then, the shared_node store happens-before the shared_node load. This load must observe the re-

claiming thread's store and the node about to be deleted cannot be accessed (shared-node-removed-safely).

Assume the dereferencing thread executed the increment first. In such a case the hazard pointer store happens-before the hazard pointer load of the reclaiming thread. Hence, the hazard pointer load must observe the dereferencing thread's store and will not delete the pointer (hp-set-safely).

13.3.5 Relaxing writes further

The dummy RMW operation will definitely synchronize a dereferencing thread and a reclaiming thread. Does this mean that other atomic operations can be relaxed? Yes, the first hazard pointer store can use relaxed_memory_order. A relaxed hazard pointer store will still synchronize-with with a load which observes that write via the RMW operation. However, the second store of 0 still need to synchronize-with the load to ensure hp-cleared-safely. Therefore, the load must still be memory_order_acquire.

Operations on shared_node are more dangerous to relax since we need to consider all shared_node accesses, not just the accesses of a reclaiming thread and a dereferencing thread. Write operations can set shared_node without using hazard pointers and load operations can read them, as long they do not dereference the pointer. Therefore, there will be no attempt to relax the shared_node operations.

13.3.6 Battling contention

Now, using the same dummy variable for all hazard pointers would make this memory location a source of contention. Much like the possible contention of the SC order we tried to avoid in the first place. The solution is to add one synchronizing dummy variable *per* thread or per hazard pointer. This removes all collisions and contention when threads are only setting hazard pointers, in fact only setting hazard pointers will not introduce any synchronization at all. When a thread reclaims data, it increments the dummy variable of ALL other threads. This is $O(N)$ operations, but since we are reclaiming at least $O(N)$ objects, the cost per object is constant.

The example implementation has one synchronizing dummy variable per hazard pointer, called synchronization_point_.

13.3.7 API summary

The API consists of three functions:

T* declare_hazard(**unsigned int** hazard_pointer_nr, **const** std::atomic<T*>& hazardous_data)
: declare that a shared pointer is about to be dereferenced.

void declare_no_hazard(**unsigned int** hazard_pointer_nr)
: declares that no more dereferencing will occur.

void retire_hazard(T* hazardous_data)

reclaim data.

The API is documented in more detail in the actual code in appendix B.

13.3.8 Implementation overview

In C++, templated code needs to reside in header files while non template code can be put into .cpp files which are compiled separately. Because of this, parts of the implementation is in a header file, found in appendix B, and part of it is in a .cpp file, found in appendix C. The header file contains the while-loop described in listing 13.3 and declarations of two methods, `detail::declare_hazard` and `detail::retire_hazard`, which are implemented in the cpp file.

The implementation is based on the abstract algorithm found in [34]. The most interesting parts of this algorithm, from a C++0x memory model point of view, has already been discussed. Rather than explaining the hazard pointers algorithm again, all variables and methods have been commented with "'a.k.a. X"' to refer to the names in original paper which can be found at

http://www.research.ibm.com/people/m/michael/ieeetpds-2004.pdf.

This way, interested readers can follow the original paper and see how it maps to C++0x by looking at the code. The implementation is intended to be structured, documented and named so that it can be easily understood by reading the code.

13.4 Stack and queue example

To illustrate lock-free programming in C++0x, the stack example of §4.3.3, part 1, and a lock-free queue based on a well known algorithm [36] are implemented. The hazard pointers implementation is used to manage lock-free memory reclamation. The examples are not really lock-free since it uses the default new allocator which is probably blocking.

The stack example is rather identical to the one explained in §4.3.3, part 1, and is not explained in further detail. The queue algorithm is a rather direct translation of the original algorithm, which is available on the web at

http://www.cs.rochester.edu/u/michael/PODC96.html.

It serves as an example of lock-free helping (see §4.3.6, part 1) and using two simultaneous hazard pointers. No memory model analyses or correctness proofs are attempted. The implementations can be found in appendix D and E.

Chapter 14

Conclusion

C++ provides a very, very rich API for expressing consistency constraints. Java-style sequential consistency is mixed with message-style acquire-release consistency and relaxed operations. Consume consistency and explicit fences are also provided. All under the same roof. Each one of these programming models is difficult to grasp. Understanding how they can be combined raises the bar even further. C++0x tries to uphold the shared memory illusion (see §2.4.2, part 1) while allowing the programmer to get close to the machine. And the machine can consist of quite different CMP architectures. Seen in the light of all of this, the complexity is not surprising.

The memory model is clearly aimed at expert library developers. The goal is to allow portable, high performance library thread-based code. If the specification is successful, C++0x could also become a platform to explore the design space of parallel programming models. At least those programming models that can be built on top of shared memory and threads. C++ is well suited for implementing embedded domain specific languages using expressive templates and generic programming. Naturally, the runtime or virtual machines of standalone languages supporting parallelism can also be implemented in C++0x.

For the memory model specification to be successful it should not contain any ambiguities or contradictions. It needs to be clearly understood by both compiler vendors and the expert developers who will use it. Even mathematically defined academic memory models contain subtle details which lead to surprising results and bugs. C++0x is quite far from an academic language so there is definitely a risk of errors and unwanted implications of the specification.

The specification should also be efficiently implementable on most mainstream CMP at the time C++0x is deployed, even those who are not released while the standard is being worked on. And CMP architectures seem to be advancing rapidly.

To my knowledge, no memory model which is nearly as advanced and targeting so different CMPs has ever been attempted. Only time will tell if it is successful.

Chapter 15

Acknowledgments

I'd like to thank Marina Papatriantafilou for doing a good job as my tutor and taking time out of her busy schedule. I want to thank Anthony Williams for his endless patience with my memory model questions and feedback, without him the second part could not have been written. I thank bwin Games, my employer, for giving me leave of absence to pursue this project. I want to thank everybody who has and will take time to read this thesis and provide feedback, it is really appreciated. Finally I'd like to Annelie and Milou for their support and patience.

Appendix A

atomic_api.hpp

```
//  Copyright Johan Torp 2008. Use, modification and distribution is subject to
//  Boost Software License, Version 1.0. (http://www.boost.org/LICENSE_1_0.txt)
//
//  Acknowledgement: Pointers and example code on how to implement the atomic
//                   API on top of Visual C++ came from Anthony Williams.
//
//  This implementation relies on Visual C++'s special semantics of volatile
//  variables (volatile can not be used for portable multithreaded C++ code),
//  a compiler fence (_ReadWriteBarrier) and some Windows OS specific
// RMW operations (starting with Interlocked)

#ifndef JOHAN_TORP_ATOMIC_HPP
#define JOHAN_TORP_ATOMIC_HPP

#include <windows.h>
#include <assert.h>
#include <boost/noncopyable.hpp>
#include <boost/type_traits.hpp>
#include <boost/utility/enable_if.hpp>


extern "C" void _ReadWriteBarrier();
#pragma intrinsic(_ReadWriteBarrier)

namespace std
{

template<typename T> class atomic;

namespace detail
{
// ------------------------------------------------------------------------------
template<class T>
inline T load_relaxed(volatile T& object)
{
  T tmp = object;
  return tmp;
}
```

```
// ------------------------------------------------------------------------
template<class T, class U>
inline void store_relaxed(volatile T& object, U new_value)
{
  object = new_value;
}
// ------------------------------------------------------------------------
template<class T>
inline T load_acquire(volatile T& object)
{
  T tmp = object;
  _ReadWriteBarrier();
  return tmp;
}
// ------------------------------------------------------------------------
template<class T, class U>
inline void store_release(volatile T& object, U new_value)
{
  _ReadWriteBarrier();
  object = new_value;
}
// ------------------------------------------------------------------------
template<class T>
inline T load_seq_cst(volatile T& object)
{
  _ReadWriteBarrier();
  T tmp = object;
  _ReadWriteBarrier();
  return tmp;
}
// ------------------------------------------------------------------------
template<class T, class U>
inline void store_seq_cst(volatile T& object, U new_value)
{
  _ReadWriteBarrier();
  object = new_value;
  _ReadWriteBarrier();
}
// ------------------------------------------------------------------------
template<typename T,typename V>
inline bool compare_and_swap_value_seq_cst(
  T volatile& target, V& comparand, T new_value)
{
  assert(sizeof(T)==sizeof(LONG));
  assert(sizeof(V)==sizeof(LONG));

  LONG const old_value = InterlockedCompareExchange(
    reinterpret_cast<LONG volatile*>(&target), new_value, comparand);
  bool succeeded = (old_value == comparand);
  comparand = old_value;

  return succeeded;
}
// ------------------------------------------------------------------------
template<typename T,typename V>
inline bool compare_and_swap_ptr_seq_cst(
  T* volatile& target, V*& comparand, T* new_value)
```

```
{
  PVOID const old_value = InterlockedCompareExchangePointer(
    reinterpret_cast<volatile PVOID*>(&target), new_value, comparand);
  bool succeeded = (old_value == comparand);
  comparand = reinterpret_cast<V*>(old_value);

  return succeeded;
}
// ---------------------------------------------------------------------------
template<typename T, typename V>
inline T fecth_and_add_seq_cst(T volatile& target, V value_to_add)
{
    assert(sizeof(T)==sizeof(LONG));
    assert(sizeof(V)==sizeof(LONG));

    return InterlockedExchangeAdd (reinterpret_cast<LONG volatile*>(&target),
                                   value_to_add);

}
// ---------------------------------------------------------------------------
/// To allow free functions friend access to member variables
template<typename T>
volatile T& get_value_intrusive(atomic<T>& atomic_object)
{
  return atomic_object.value_;
}
// ---------------------------------------------------------------------------
} // end namespace detail
// ---------------------------------------------------------------------------
enum memory_order
{
    memory_order_relaxed, memory_order_acquire, memory_order_release,
    memory_order_acq_rel, memory_order_seq_cst
};
// ---------------------------------------------------------------------------

/// This class does not correspond to the std::atomic template for
/// user defined types.
template<typename T>
class atomic : private boost::noncopyable
{
public :
// ---------------------------------------------------------------------------
  atomic(T value)
  : value_(value) // Writing to volatile, same as relaxed write
  {}
// ---------------------------------------------------------------------------
  T load(memory_order order=memory_order_seq_cst) volatile const
  {
    switch (order)
    {
      case memory_order_relaxed:
        return detail::load_relaxed(value_);
      case memory_order_acquire:
        return detail::load_acquire(value_);
      case memory_order_seq_cst:
        return detail::load_seq_cst(value_);
```

```
    }
    assert(!"Illegal memory order for load");
    return value_;
  }
// ----------------------------------------------------------------------
  void store(T value, memory_order order=memory_order_seq_cst) volatile
  {
    switch (order)
    {
      case memory_order_relaxed:
        return detail::store_relaxed(value_, value);
      case memory_order_release:
        return detail::store_release(value_, value);
      case memory_order_seq_cst:
        return detail::store_seq_cst(value_, value);
    }
    assert(!"Illegal memory order for store");
  }
// ----------------------------------------------------------------------
private:
  template<class T>
  friend volatile T& detail::get_value_intrusive(atomic<T>& atomic_object);

  volatile T value_;
};
// ----------------------------------------------------------------------
// SFINAE specialization for integer types
template<class T, class V>
inline
typename boost::enable_if_c<boost::is_integral<T>::value, T>::type
atomic_fetch_add_explicit(
  atomic<T>* atomic_object, V value_to_add, memory_order order)
{
  // Ignore memory order, map to seq cst implementation
  return detail::fecth_and_add_seq_cst(
           get_value_intrusive(*atomic_object), value_to_add);
}
// ----------------------------------------------------------------------
// SFINAE specialization for non-pointer types
// Comparand will be updated with the old value if CAS fails
template<class T, class V>
inline
typename boost::enable_if_c<!boost::is_pointer<T>::value, bool>::type
atomic_compare_exchange(atomic<T>* atomic_object,
                        V* comparand,
                        T new_value)
{
  return detail::compare_and_swap_value_seq_cst(
           get_value_intrusive(*atomic_object), *comparand, new_value);
}
// ----------------------------------------------------------------------
// SFINAE specialization for pointers
// Comparand will be updated with the old value if CAS fails
template<class T, class V>
inline
typename boost::enable_if_c<boost::is_pointer<T>::value, bool>::type
atomic_compare_exchange(atomic<T>* atomic_object,
```

```
                          V* comparand,
                          T new_value)
{
  return detail::compare_and_swap_ptr_seq_cst(
            get_value_intrusive(*atomic_object), comparand, new_value);
}
// ------------------------------------------------------------------------
// In a real atomic API implementation, atomic_int is a first class type
typedef atomic<int> atomic_int;

} // end namespace std

#endif
```

Appendix B

hazard_pointer.hpp

```
//  Copyright Johan Torp 2008. Use, modification and distribution is subject to
//  Boost Software License, Version 1.0. (http://www.boost.org/LICENSE_1_0.txt)
//
//  There is a US patent claim on hazard pointers. Some kind of license or
//  permission might be required to use hazard pointers in the United States.
#ifndef JOHAN_TORP_HP_HPP
#define JOHAN_TORP_HP_HPP

#include "../atomic/atomic_api.hpp"

namespace jtorp
{
namespace detail
{
// ----------------------------------------------------------------------------
template<typename T>
void call_delete(void* data) { delete static_cast<T*>(data); }
// ----------------------------------------------------------------------------
typedef void (*delete_function)(void*);
// ----------------------------------------------------------------------------
void retire_hazard(void* hazardous_data, delete_function deleter);
// ----------------------------------------------------------------------------
// Won't expose this "raw" version until I need it
void declare_hazard(unsigned int hazard_pointer_nr, void* hazardous_data);
// ----------------------------------------------------------------------------
} // end namespace detail
// ----------------------------------------------------------------------------

/// Declares that your are accessing shared memory which must not be deleted
///
/// If 0 is returned, there is no need to call declare_no_hazard since a null
/// pointer can never be hazardous.
///
/// This implementation is only lock-free so the entire hazard pointer library
/// looses its wait-free guarantee.
///
/// @returns A snapshot value of the hazardous pointer which is guaranteed to
///          be visible to other threads
```

```
/// @param hazard_pointer_nr Must be 0 or 1. This library only supports
///                          two hazard pointers per thread
/// @param snapshot A pre-loaded snapshot value of the hazard pointer which
///                 must have been loaded with at least acquire semantics
template<class T>
inline T* declare_hazard(unsigned int hazard_pointer_nr,
                         const std::atomic<T*>& hazardous_data, T* snapshot)
{
  while (true)
  {
    detail::declare_hazard(hazard_pointer_nr, snapshot);
    // Test that the hazardous_data still points to our snapshot,
    // otherwise it might have been reclaimed!
    T* snapshot_after_hazard_declared =
      hazardous_data.load(std::memory_order_acquire);

    if (snapshot == snapshot_after_hazard_declared) return snapshot;

    // The test failed. Reuse read snapshot to minimize atomics calls
    snapshot = snapshot_after_hazard_declared;
  }
}
// ---------------------------------------------------------------------------

/// @see declare_hazard two-parameter version.
///      This version does not require a pre-loaded snapshot.
template<class T>
inline T* declare_hazard(unsigned int hazard_pointer_nr,
                         const std::atomic<T*>& hazardous_data)
{
  T* snapshot = hazardous_data.load(std::memory_order_acquire);
  return declare_hazard(hazard_pointer_nr, hazardous_data, snapshot);
}
// ---------------------------------------------------------------------------

/// Declare that you are no longer holding a hazardous pointer
///
/// @param hazard_pointer_nr Must be 0 or 1. This library only supports
///                          two hazard pointers per thread
inline void declare_no_hazard(unsigned int hazard_pointer_nr)
{
  detail::declare_hazard(hazard_pointer_nr, 0);
}
// ---------------------------------------------------------------------------

/// @pre declare_no_hazard must have been called before retiring a node
///
/// @param hazardous_data Might be deleted/reclaimed immediately or stored
///                       for later reclamation
template<class T>
inline void retire_hazard(T* hazardous_data)
{
  detail::retire_hazard(hazardous_data, &detail::call_delete<T>);
}
// ---------------------------------------------------------------------------
}
#endif
```

Appendix C

hazard_pointer.cpp

```
//  Copyright Johan Torp 2008. Use, modification and distribution is subject to
//  Boost Software License, Version 1.0. (http://www.boost.org/LICENSE_1_0.txt)
//
//  There is a US patent claim on hazard pointers. Some kind of license or
//  permission might be required to use hazard pointers in the United States.

#include "hazard_pointer.hpp"
#include <algorithm> // For sort and binary_search
#include <vector>
#include <boost/noncopyable.hpp>
#include <boost/thread/tss.hpp>

namespace jtorp
{

namespace
{
// ----------------------------------------------------------------------------
// A.k.a. N
const unsigned int MAX_SIMULTANEOUS_THREADS = 100;
// A.k.a. K
const unsigned int HAZARD_POINTERS_PER_THREAD = 2;
// A.k.a. H.
const unsigned int MAX_HAZARD_POINTERS = HAZARD_POINTERS_PER_THREAD*
                                         MAX_SIMULTANEOUS_THREADS;
// A.k.a.  R. Must be a factor larger than H to guarantee O(R) nodes reclaimed
// each time reclamation is tried. This is because in theory, H-1
// hazard pointers can be held and therefor not reclaimed.
const unsigned int MAX_RETIRED_NODES = 2*MAX_HAZARD_POINTERS;
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
  /// A bounded size pointer collection containing at most MAX_HAZARD_POINTERS
  class pointer_collection {
  public:
    pointer_collection() : size_(0) {}
// ----------------------------------------------------------------------------
    void push(void* ptr)
```

```
    {
      assert(size_<MAX_HAZARD_POINTERS);
      data[size_++] = ptr;
    }
// ------------------------------------------------------------------------
    void sort() { std::sort(data, data+size_); }
// ------------------------------------------------------------------------
    /// @pre sort() is the last mutating function called
    bool contains(void* ptr) const
    {
      return std::binary_search(data, data+size_, ptr);
    }
    void clear() {size_ = 0;}
// ------------------------------------------------------------------------
  private:
    void* data[MAX_HAZARD_POINTERS];
    unsigned int size_;
  };
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// bool is not used because the example atomic API implementation does not
// support CAS for bools
const int VACANT = 0;
const int IN_USE = 1;

// Helper struct to allocate an item from a shared pool.
// Does not model OO encapsulation.
// Should not be used in production code - does not adress memory locality.
template<class T, unsigned int POOL_SIZE>
struct pool {
// ------------------------------------------------------------------------
  inline bool try_allocate(T& item)
  {
    // compare_exchange will write to this variable if it fails,
    // hence the local copy
    int vacant = VACANT;

    while (true)
    {
      if (std::atomic_compare_exchange(&item.in_use_, &vacant, IN_USE))
      {
        return true;
      }
      if (vacant != VACANT) return false;
      // else, we have a spurios failure and should retry
    }
  }
// ------------------------------------------------------------------------
  T& allocate()
  {
    for (unsigned int i=0; i<POOL_SIZE;++i)
    {
      if (try_allocate(at(i))) return at(i);
    }

    abort(); return at(0); // Dummy return
```

```
  }
// ------------------------------------------------------------------------
  void release(T& item)
  {
    item.in_use_.store(VACANT, std::memory_order_release);
  }
// ------------------------------------------------------------------------
  static unsigned int size() { return POOL_SIZE; }
// ------------------------------------------------------------------------
  T& at(unsigned int element)
  {
    assert(element < POOL_SIZE);
    return pool_[element];
  }
private:
  T pool_[POOL_SIZE];
};

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------

// The hazard pointers owned by a single thread and associated data
struct hazard_pointer : private boost::noncopyable
{
  hazard_pointer() : in_use_(VACANT), data_(0), synchronization_point_(0)
{}

  std::atomic_int in_use_; // Used to claim ownership of this reusable struct
  std::atomic<void*> data_; // The actual hazard pointer
  // Dummy value used for synchronization. Mutable since it is dummy state
  mutable std::atomic_int synchronization_point_;
};

// A real world hazard pointers implemenation should be using thread local
// storage for storing the hazard pointers and retired nodes list
pool<hazard_pointer, MAX_HAZARD_POINTERS> hazard_pointer_pool;

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------

  // The retired nodes, eligble for reclamation if no other threads hold
  // hazard pointers to them. These are re-used between threads. The main
  // rationale is that running threads can help clean retired nodes of "dead"
  // threads which couldn't be reclaimed when they were exiting.
  // I call these retired nodes of dead threads for DEAD NODES.
class retired_nodes
{
public:
  retired_nodes() : in_use_(VACANT), size_(0)  {}

  std::atomic_int in_use_; // Exposed member variable
// ------------------------------------------------------------------------
  void push(void* data, detail::delete_function deleter)
  {
    assert(!full());
```

```
    list_[size_].data_ = data;
    list_[size_].deleter_ = deleter;

    ++size_;
  }
// ------------------------------------------------------------------------
  void reclaim_nodes()
  {
    scan_for_hazardous_data(); // Build up hazardous_data_list_, a.k.a stage 1
    reclaim_safe_nodes(); // Perform reclamation, a.k.a stage 2
  }
// ------------------------------------------------------------------------
  bool full() const { return size_ == MAX_RETIRED_NODES;}
// ------------------------------------------------------------------------
private:

  // Step 1 of reclamation. Clear, build up and sort hazardous_data_list_
  // (a.k.a. plist). This list contains hazardous data nodes which are being
  // accessed by other threads
  void scan_for_hazardous_data()
  {
    hazardous_data_list_.clear();

    for (unsigned int i=0 ; i<hazard_pointer_pool.size(); ++i)
    {
      hazard_pointer& hp = hazard_pointer_pool.at(i);

      // See §13.3.4
      std::atomic_fetch_add_explicit(&hp.synchronization_point_, 1,
                                     std::memory_order_acq_rel);

      if (void* hazardous_data = hp.data_.load(std::memory_order_acquire))
      {
        hazardous_data_list_.push(hazardous_data);
      }
    }
    hazardous_data_list_.sort();
  }
// ------------------------------------------------------------------------
  void reclaim_safe_nodes()
  {
    const unsigned int NODES_TO_RECLAIM = size_;
    size_ = 0;
    for (unsigned int i = 0; i<NODES_TO_RECLAIM;++i)
    {
      node& n = list_[i];
      if (hazardous_data_list_.contains(n.data_))
      { // Reclaim later
        list_[size_++] = n;
      }
      else
      { // Reclaim now
        n.deleter_(n.data_);
      }
    }
  }
```

```
// ------------------------------------------------------------------------
  struct node
  {
    void* data_;
    detail::delete_function deleter_;
  };

  node list_[MAX_RETIRED_NODES]; // A.k.a. rlist
  unsigned int size_; // A.k.a. rcount
  pointer_collection hazardous_data_list_; // A.k.a. plist
};

// A realworld hazard pointers implemenation should be using thread local
// storage for storing the hazard pointers and retired nodes list
pool<retired_nodes, MAX_SIMULTANEOUS_THREADS> retired_nodes_pool;

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------


class hazard_manager {
public:
// ------------------------------------------------------------------------
  hazard_manager()
  : retired_nodes_(retired_nodes_pool.allocate())
  {
    for (unsigned int i = 0; i < HAZARD_POINTERS_PER_THREAD;++i)
    {
      hazard_pointers_.push_back(&hazard_pointer_pool.allocate());
    }
  }
// ------------------------------------------------------------------------
  // C++0x thread local storage supports non-trivial destructors.
  // In an imagined C++0x program, this code should be run at thread exit.
  // There is also a std::at_thread_exit mechanism.
  ~hazard_manager()
  {
    for (unsigned int i = 0; i < HAZARD_POINTERS_PER_THREAD;++i)
    {
      set_hazard(i, 0); // Don't leave a dangling hazard pointer
      // Return hazard pointer to global pool
      hazard_pointer_pool.release(*hazard_pointers_.at(0));
    }
    // Hand over remaining retired but unreclaimed nodes to global pool
    retired_nodes_pool.release(retired_nodes_);
    help_reclaim_dead_nodes(); // Try cleaning items in global pool
  }
// ------------------------------------------------------------------------
  void retire_hazard(void* hazardous_data, detail::delete_function deleter)
  {
    retired_nodes_.push(hazardous_data, deleter);

    // If retired nodes are full, scan and reclaim nodes (a.k.a scan).
    // Make sure we reclaim at least one node so we don't push the next node
    // at a full container. If R>H - as it should - this is guaranteed to be
    // the case. The while loop is just an extra safety net
```

```
    while (retired_nodes_.full()) retired_nodes_.reclaim_nodes();
  }
// ------------------------------------------------------------------------
  void set_hazard(unsigned int hazard_pointer_nr, void* hazardous_data)
  {
    hazard_pointer& hp = *hazard_pointers_[hazard_pointer_nr];

    if (hazardous_data != 0)
    {
      hp.data_.store(hazardous_data, std::memory_order_relaxed);
      // See §13.3.4
      std::atomic_fetch_add_explicit(&hp.synchronization_point_, 1,
                                     std::memory_order_acq_rel);
    } else {
      hp.data_.store(0, std::memory_order_release);
    }
  }
// ------------------------------------------------------------------------
private:
// ------------------------------------------------------------------------
  void help_reclaim_dead_nodes()
  {
    for (unsigned int i=0; i<retired_nodes_pool.size() ; ++i)
    {
      retired_nodes& list = retired_nodes_pool.at(i);
      if (retired_nodes_pool.try_allocate(list))
      {
        list.reclaim_nodes();
        retired_nodes_pool.release(list);
      }
    }
  }
// ------------------------------------------------------------------------
  std::vector<hazard_pointer*> hazard_pointers_;
  retired_nodes& retired_nodes_; // A.k.a. rlist and rcount
}; // end class hazard_manager

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
 // Visual C++ workaround since __declspec(thread) doesn't support
 // non-trivial construction or destructors
boost::thread_specific_ptr<hazard_manager> hazard_manager_ptr;

hazard_manager& get_my_manager()
{
  if (hazard_manager_ptr.get() == 0)
  {
    hazard_manager_ptr.reset(new hazard_manager);
  }
  return *hazard_manager_ptr;
}

} // end anonymous namespace

namespace detail
{
```

```
// ------------------------------------------------------------------------
void retire_hazard(void* hazardous_data, detail::delete_function deleter)
{
  get_my_manager().retire_hazard(hazardous_data, deleter);
}
// ------------------------------------------------------------------------
void declare_hazard(unsigned int hazard_pointer_nr, void* hazardous_data)
{
  assert(hazard_pointer_nr < HAZARD_POINTERS_PER_THREAD);
  get_my_manager().set_hazard(hazard_pointer_nr, hazardous_data);
}
// ------------------------------------------------------------------------
} // end namespace detail
} // end namespace jtorp
```

Appendix D

stack.hpp

```
//  Copyright Johan Torp 2008. Use, modification and distribution is subject to
//  Boost Software License, Version 1.0. (http://www.boost.org/LICENSE_1_0.txt)
#ifndef JOHAN_TORP_STACK_HPP
#define JOHAN_TORP_STACK_HPP

#include "../hazard_pointer/hazard_pointer.hpp"
#include <memory> // For auto_ptr
#include <boost/noncopyable.hpp>

namespace jtorp
{

template<typename T>
class stack : private boost::noncopyable
{
public:
// ------------------------------------------------------------------------------
  stack() : head_(0) {}
// ------------------------------------------------------------------------------
  ~stack()
  {
    while (!empty()) pop();
  }
// ------------------------------------------------------------------------------
  void push(const T& data)
  {
    // Using auto_ptr in case some code can throw an exception.
    // TODO: Investigare if that can happen, otherwise change to raw pointer
    std::auto_ptr<node> new_head(new node);
    new_head->data_.reset(new T(data));

    // Repoint new_head->next_ to old head
    new_head->next_ = head_.load(std::memory_order_acquire);

    // CAS reloads new_node->next_ on failure,
    // we use this fact to avoid extra atomic_load calls
    while (!atomic_compare_exchange(&head_, &new_head->next_, new_head.get()));
```

```
    new_head.release(); // new_node will be reclaimed in pop
  }
// ------------------------------------------------------------------------

  /// Returns invalid ptr if stack is empty
  std::auto_ptr<T> pop()
  {
    // Pre-load head to use the efficient version of declare_no_hazard
    node* old_head_ = head_.load(std::memory_order_acquire);

    // Pure optimization for empty stack case,
    // still needs to be rechecked below
    if (old_head_ == 0) return std::auto_ptr<T>();

    while (true)
    {
      old_head_ = declare_hazard(0, head_, old_head_);
      if (old_head_ == 0)
      {
        // No need to declare_no_hazard since declare hazard returned 0
        return std::auto_ptr<T>();
      }
      // compare_exchange will update old_head if the CAS fails.
      // This is used in declare_hazard above to avoid extra atomic load
      if (std::atomic_compare_exchange(&head_, &old_head_, old_head_->next_)) {
        break;
      }
    }

    std::auto_ptr<T> old_data(old_head_->data_);
    declare_no_hazard(0);
    retire_hazard(old_head_);

    return old_data;
  }
// ------------------------------------------------------------------------
  bool empty() const { return head_.load(std::memory_order_acquire) == 0; }
// ------------------------------------------------------------------------
private:
// ------------------------------------------------------------------------
  struct node : private boost::noncopyable
  {
    node* next_;
    std::auto_ptr<T> data_;
  };

  std::atomic<node*> head_;
};

}

#endif
```

Appendix E

queue.hpp

```
//  Copyright Johan Torp 2008. Use, modification and distribution is subject to
//  Boost Software License, Version 1.0. (http://www.boost.org/LICENSE_1_0.txt)
#ifndef JOHAN_TORP_QUEUE_HPP
#define JOHAN_TORP_QUEUE_HPP

#include "../hazard_pointer/hazard_pointer.hpp"
#include <memory> // For auto_ptr
#include <boost/noncopyable.hpp>

namespace jtorp
{

template<typename T>
class queue : private boost::noncopyable {
public:
// ------------------------------------------------------------------------------
  queue()
  : head_(new node(0)), // Sentinel node with data ptr == 0
    tail_(head_.load(std::memory_order_relaxed)) {}
// ------------------------------------------------------------------------------
  ~queue()
  {
    node* head = head_.load(std::memory_order_acquire);
    while (tail_.load(std::memory_order_acquire) != head) pop_front();
    delete head; // Manually delete sentinel node
  }
// ------------------------------------------------------------------------------
  void push_back(const T& data)
  {
    // push_back is done in two steps. First we repoint tail->next_ to
    // our new node. This is the linearization point
    // Secondly we repoint tail to new node. If we fail the first step we must
    // help out with the second in case the thread which succeeded at the
    // linearization point has been swapped out

    node* new_node = new node(new T(data));

    while (true)
```

```
{
  node* tail_snapshot = declare_hazard(0, tail_);
  std::atomic<node*>& tail_next_ptr_ref = tail_snapshot->next_;
  node* expected_end = 0;

  while (true) // While we have spurios failures from the first CAS
  {
    // The linearization point, re-point tail->next_ to new_node. If this
    // call fails, expected_end is updated to point to the new tail.
    bool insertion_step_1_successful =
      std::atomic_compare_exchange(&tail_next_ptr_ref,
                                   &expected_end, new_node);

    // Check for spurios failure of above CAS. In the normal case,
    // expected end should not remain zero if the CAS failed
    if (!insertion_step_1_successful && expected_end == 0) continue;

    // We will no longer access a dereferenced part (tail_next_ptr_ref)
    // of our tail_ snapshot
    declare_no_hazard(0);

    if (insertion_step_1_successful)
    {

      // If we get swapped out at this point we would block all other
      // threads unless they helped us finish
      bool spurios_failure = false;
      do
      {
        node* tail_snapshot_copy = tail_snapshot;
        // Re-point tail to new node unless another thread helped us do so
        spurios_failure =
          !std::atomic_compare_exchange(&tail_, &tail_snapshot_copy,
                                        new_node)
          && tail_snapshot_copy == tail_snapshot;
      } while (spurios_failure);
      // At this point, either we or some other thread has re-pointed tail
      // to new_node and we are done
      return;
    }
    else
    {
      assert(expected_end != 0);
      // Some other thread succeeded in inserting it's element.
      // expected_end is the new tail. Help that thread with it's second
      // step to ensure lock-free property (it might have been swapped out)
      std::atomic_compare_exchange(&tail_, &tail_snapshot, expected_end);
      // We ignore spurios failures here. If we get one and the succeeding
      // thread really is swapped out we will come here again and try
      // helping again. This costs some atomic operations but spurios
      // failures are expected to be really rare

      break; // Skip to outer loop
    }
  }
}
```

```
  }
// -----------------------------------------------------------------------

  /// Returns null ptr if queue is empty
  std::auto_ptr<T> pop_front()
  {
    while (true)
    {
      node* tail_snapshot = tail_.load(std::memory_order_acquire);
      node* head_snapshot = declare_hazard(0, head_);
      node* head_next_snapshot = declare_hazard(1, head_snapshot->next_);
      T* potential_data = head_next_snapshot != 0
                        ? head_next_snapshot->data_
                        : 0;
      declare_no_hazard(0);
      declare_no_hazard(1);

      if (potential_data == 0) return std::auto_ptr<T>();


      if (head_snapshot == tail_snapshot)
      {
        // Another thread is half-finished enqueing the only item in the stack.
        // Help it, ignore outcome and then retry dequeing
        std::atomic_compare_exchange(&tail_, &tail_snapshot,
                                     head_next_snapshot);
      }
      else// The head item exists and is fully enqueued
      {
        // Linearization point, try popping head
        if (std::atomic_compare_exchange(&head_, &head_snapshot,
                                         head_next_snapshot))
        {
          retire_hazard(head_snapshot);
          return std::auto_ptr<T>(potential_data);
        }
        // Either a spurios failure or someone else dequeued head.
        // Retry either way
      }
    }
  }
// -----------------------------------------------------------------------
private:
  struct node : private boost::noncopyable
  {
    node(T* data) : next_(0), data_(data) {}
    std::atomic<node*> next_;
    T* data_;
  };

  std::atomic<node*> head_;
  std::atomic<node*> tail_;
}; // end class queue

}
#endif
```

Bibliography

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, 1994.

[3] Pete Becker. Working draft, c++ programming language. mailings N2691, C++ standards committee, Jun 2008.

[4] Hans-J. Boehm. Google tech talk: Getting c++ threads right. Publication at http://video.google.com/videoplay?docid=-7297026930630041154.

[5] Hans-J. Boehm. Why undefined semantics for c++ data races? Publication at http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/why_undef.html.

[6] Hans-J. Boehm. Sequential consistency for atomics. mailings N2177, C++ standards committee, Mar 2007.

[7] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *PLDI*, pages 157–164, 1991.

[8] Hans-J. Boehm Clark Nelson. Concurrency memory model (final revision). mailings N2429, C++ standards committee, Oct 2007.

[9] Lawrence Crowl. Strong compare and exchange. mailings N2748, C++ standards committee, Aug 2008.

[10] SOA World Magazine News Desk. Moore's law: "we see no end in sight," says intel's pat gelsinger. Publication at http://java.sys-con.com/node/557154, May 2008.

[11] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. *SIGPLAN Not.*, 38(2 supplement):163–174, 2003.

[12] Manek Dubash. Moore's law is dead, says gordon moore. Publication at http://www.techworld.com/opsys/news/index.cfm?NewsID=3477, Apr 2005.

[13] EDN. Intel, amd diverge on multicore strategies. Publication at http://www.edn.com/article/CA6445861.html, May 2007.

[14] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18(3a):15–26, 1990.

[15] Anders Gidenstam, Marina Papatriantafilou, Hakan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *ispan*, 00:202–207, 2005.

[16] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Allocating memory in a lock-free manner. In *ESA*, pages 329–342, 2005.

[17] M. Girkar and C. D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):166–178, 1992.

[18] Darien Graham-Smith. Larrabee "like a gpu from 2006". Publication at http://www.pcpro.co.uk/news/220947/nvision-larrabee-like-a-gpu-from-2006.html, Aug 2008.

[19] Dan Grossman. The transactional memory / garbage collection analogy. *SIGPLAN Not.*, 42(10):695–706, 2007.

[20] Lawrence Crowl Hans-J. Boehm. C++ atomic types and operations. mailings N2427, C++ standards committee, Oct 2007.

[21] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, 2003.

[22] T. E. Hart, P. E. McKenney, and A. D. Brown. Making lockless synchronization fast: Performance implications of memory reclamation. In *Proc. IEEE International Parallel & Distributed Processing Symposium (20th IPDPS'06)*, Rhodes Island, Greece, April 2006. IEEE Computer Society.

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

[24] IBM. Cell broadband engine interconnect and memory interface. Publication at http://www.hotchips.org/archives/hc17/2_Mon/HC17.S1/HC17.S1T2.pdf, 2005.

[25] Intel. Larrabee: A many-core x86 architecture for visual computing. Publication at http://softwarecommunity.intel.com/UserFiles/en-us/File/larrabee_manycore.pdf, Aug 2008.

[26] William Pugh Jeremy Manson. The java memory model. Technical report, University of Illinois, Jan 2005.

[27] Vigyan Kaushik. Tips for writing efficient sql queries. Publication at http://www.dbapool.com/downloads/whitepapers/tipsforesql.pdf.

[28] Tom Knight. An architecture for mostly functional languages. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112, New York, NY, USA, 1986. ACM.

[29] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[30] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[31] Leslie Lamport. On interprocess communication. In *Distributed Computing 1*, pages 77–101, 1986.

[32] Jeremy Manson and Brian Goetz. Jsr 133 (java memory model) faq. Publication at http://www.cs.umd.edu/ pugh/java/memoryModel/jsr-133-faq.html, 2004.

[33] Sally A. McKee. Reflections on the memory wall. In Stamatis Vassiliadis, Jean-Luc Gaudiot, and Vincenzo Piuri, editors, *Conf. Computing Frontiers*, page 162. ACM, 2004.

[34] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[35] Maged M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, 39(6):35–46, 2004.

[36] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[37] Jay Hoeflinger Michael Suess. An interview with dr. jay hoeflinger about automatic parallelization. Publication at http://www.thinkingparallel.com/2007/08/14/an-interview-with-dr-jay-hoeflinger-about-automatic-parallelization/.

[38] Microsoft. The manycore shift white paper. Publication at http://www.microsoft.com/presspass/events/supercomputing/docs/ManycoreWP.doc, Nov 2007.

[39] MSDN. Thread stack size. Publication at http://msdn.microsoft.com/en-us/library/ms686774(VS.85).aspx, Jul 2008.

[40] Richard Nass, James Reinders Dr. James Truchard, and Herb Sutter Edward A. Lee. Editorial webinar: Software strategies for embedded multicore development. Publication at http://www.techonline.com/learning/webinar/208400266, Apr 2005.

[41] Eric Niebler. Trip report: Ad-hoc meeting on threads in c++. Publication at http://www.artima.com/cppsource/threads_meeting.html.

[42] Lawrence Crowl Paul E. McKenney, Hans-J. Boehm. C++ data-dependency ordering: Atomics and memory model. mailings N2664, C++ standards committee, Jun 2008.

[43] William Pugh. The java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6):445–455, 2000.

[44] James Reinders. Are multicore processors here to stay? Publication at http://www.devx.com/go-parallel/Article/36003/1763, Jul 2008.

[45] Intel Tera Scale. Tera scale web page. Publication at http://techresearch.intel.com/articles/Tera-Scale/1421.htm, August 2008.

[46] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *ISMM '06: Proceedings of the 5th international symposium on Memory management*, pages 84–94, New York, NY, USA, 2006. ACM.

[47] Herb Sutter. Prism draft 0.8. Publication at http://www.gotw.ca/memmodel/Prism - draft 0.8.pdf, Jul 2006.

[48] Herb Sutter. The concurrency landrush. Doctor Dobb's Journal. Publication at http://herbsutter.spaces.live.com/Blog/cns!2D4327CC297151BB!362.entry, Dec 2007.

[49] Herb Sutter. Use lock hierarchies to avoid deadlock. Doctor Dobb's Journal. Publication at http://www.ddj.com/hpc-high-performance-computing/204801163, December 2007.

[50] Herb Sutter. Maximize locality, minimize contention. Doctor Dobb's Journal. Publication at http://www.ddj.com/architect/208200273, May 2008.

[51] Simon Peyton Jones Tim Harris, Simon Marlow and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, 2008.

[52] HPC Wire. Amd sees a heterogeneous future. Publication at http://www.hpcwire.com/features/17897699.html, February 2007.