

Finding parallelism How to survive in a multi-core world

Presented to the Stuttgart Media University in fulfillment of the thesis requirement for the degree of Bachelor of Science in Computer Science and Media

By Kai Jäger

Supervisors:
Prof. Walter Kriha - Stuttgart Media University
Claus Gittinger – eXept Software AG

Stuttgart
August 25, 2008

Revision 1, September 4, 2008

Finding parallelism

How to survive in a multi-core world

by Kai Jäger

Abstract

With multi-core CPUs in every new PC and many-core CPUs on the horizon, it becomes increasingly apparent that we are standing on the verge of a new era. Multi-core promises vast performance resources, but harnessing those resources is no longer free. To exploit the power of multi-core, we need to write parallel software. But in the absence of parallel hardware, there has been little motivation in the past for developers to concern themselves with parallel programming. Consequently, this sudden paradigm shift is often met with fear and skepticism. At the same time, programming language- and library designers are just now beginning to embrace this new trend, while developers are often stuck with inadequate pre-multi-core languages and libraries.

This thesis hopes to offer an insight into what the multi-core trend implies. It describes the most common issues with parallel programming and how certain programming languages address them. Furthermore, it discusses the different models of parallel programming and presents their benefits and downsides.

Statement of originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices.

Kai Jäger, August 2008

Acknowledgements

Nothing is ever written in a void and this thesis is no exception. I would like to use this opportunity to thank Prof. Walter Kriha for the inspiration and the support, as well as Claus Gittinger for sparking my interest in programming languages. I would also like to thank Herb Sutter, whose excellent article “The free lunch is over” [1] has opened my eyes about the implications of the multi-core revolution. While doing the research for this thesis, I contacted a number of international experts on various issues and received an overwhelming number of responses. I would therefore like to express my utmost gratitude towards Prof. David. G Myers of Hope College, who helped me understand, why it is so difficult for us as humans to reason about parallel programs. I would also like to thank Guido van Rossum, Dr. Bjarne Stroustrup, Joe Armstrong, Simon Peyton-Jones, Larry Wall, Yukihiro Matsumoto and Roberto Ierusalimschy, all of who were kind enough to explain to me their perspective on how the multi-core revolution will affect programming languages (see Appendix). Lastly I would like to thank my good friend Tim Pehrson, who gave me important feedback on my writing style and who has been my primary connection to the US for the past ten years.

Contents

1. Introduction	1
1.1. The end of rising clock-speeds	1
1.2. Exploiting the power of multi-core	2
1.3. Languages for parallel programming	4
1.4. The human factor	6
2. Concurrent programming today	7
2.1. Threads and shared state	7
2.2. Race conditions	7
2.3. Locks	9
2.3.1. Mutual exclusion	10
2.3.2. Locking granularity	11
2.3.3. Composability	12
2.3.4. Deadlocks	13
2.4. Correctness	15
3. Alternative approaches to concurrency	17
3.1. Software transactional memory	17
3.1.1. Shared-state concurrency revisited	17
3.1.2. Memory transactions	18
3.1.3. Inside software transactional memory	19
3.1.4. Condition variables	21
3.1.5. Exception safety	23
3.1.6. Limitations	23
3.1.7. Performance	24
3.1.8. The garbage collection analogy	24
3.1.9. Conclusion	25
3.2. Message passing concurrency	26

3.2.1. Actors and messages	27
3.2.2. Programming with messages.....	28
3.2.3. Fault tolerance.....	29
3.2.4. Scalability	30
3.2.5. Conclusion	31
4. Finding parallelism	32
4.1. Amdahl's law.....	32
4.2. Data parallelism	33
4.2.1. Automatic parallelization	33
4.3. Task parallelism	35
4.4. Scalability	36
4.5. Running out of parallelism.....	36
5. Conclusions.....	38
References.....	40
Appendix.....	42

List of figures

Figure 1: A comparison of the clock speeds and transistor counts of Intel CPUs...	2
Figure 2: The Transfer procedure.....	8
Figure 3: The Withdraw procedure	8
Figure 4: A sequence diagram illustrating the lost update problem	9
Figure 5: Locking applied to the Transfer example.....	10
Figure 6: Implementations of the Lock and Unlock functions.....	10
Figure 7: An exemplary implementation of test-and-set.....	11
Figure 8: The thread-safe Withdraw function.....	12
Figure 9: Taking the two locks in the Transfer function	13
Figure 10: A typical deadlock situation	14
Figure 11: Sequence diagram of a typical deadlock.....	14
Figure 12: Ordered lock acquisition	15
Figure 13: The transfer function using software transactional memory.....	19
Figure 14: A memory transaction dissected.....	20
Figure 15: An exemplary implementation of compare-and-swap	21
Figure 16: A blocking version of the Withdraw function	22
Figure 17: The bank account example implemented using the actor model	28
Figure 18: Sending messages to the account actor.....	29
Figure 19: Performance gain from parallelizing parts of an application.....	32
Figure 20: A function with the “pure” annotation	34
Figure 21: The parallelized loop.....	34

1. Introduction

Today, most consumer PCs and notebooks are equipped with multi-core CPUs. What sounds like yet another marketing buzz-word may very well mark the beginning of a new era. In this chapter, we will look at how the multi-core revolution came to be, what its effects are on the way we are going to write software in the future and how it relates to programming languages.

1.1. The end of rising clock-speeds

The multi-core revolution has caught many people by surprise. We have gotten so used to the fact that CPUs “magically” keep becoming faster, that we have completely neglected the possibility that this trend could come to an end. Moore’s law predicts exponential growth for the complexity of integrated circuits. For the longest time, this phenomenon could be witnessed in the ever rising clock-speeds of modern CPUs. But while Moore’s law¹ still holds true, the rise in clock-speeds has begun to stagnate several years ago. Around 2002, clock-speeds had become so fast that it was no longer possible to reach all the crevices of a chip in a single cycle [1]. CPU manufacturers have since tried to hide that latency from the software by exploiting instruction level parallelism². This effort has recently brought us CPUs just shy of 4 GHz native clock-speed, but whether we will see any additional increase in clock-speed remains uncertain.

Increasing clock-speeds have been the driving force for many developments in the software world. However, CPUs that operate at such high clock-speeds have two undesirable qualities: they consume a lot of energy and they produce a lot of heat. Both of these effects could be prevented by operating the CPUs at lower clock-speeds, but sacrificing processing power for the sake of better energy-efficiency may not be acceptable in many usage scenarios. By putting multiple CPU cores onto a single chip however, clock-speeds can be reduced while main-

¹ Moore’s law, named after Dr. Gordon Moore who is a co-founder of Intel, predicts that the complexity of integrated circuits (and thus CPUs) doubles every 24 months.

² Instruction level parallelism is a term that describes the ability of a CPU to execute several operations in parallel. This includes measures like out-of-order execution, branch prediction and instruction pipelining.

taining or even exceeding the performance of a single-core CPU running at a much higher clock-speed. This is the road that most CPU manufacturers have taken and it is the reality that all of us who work in software will have to face sooner or later.

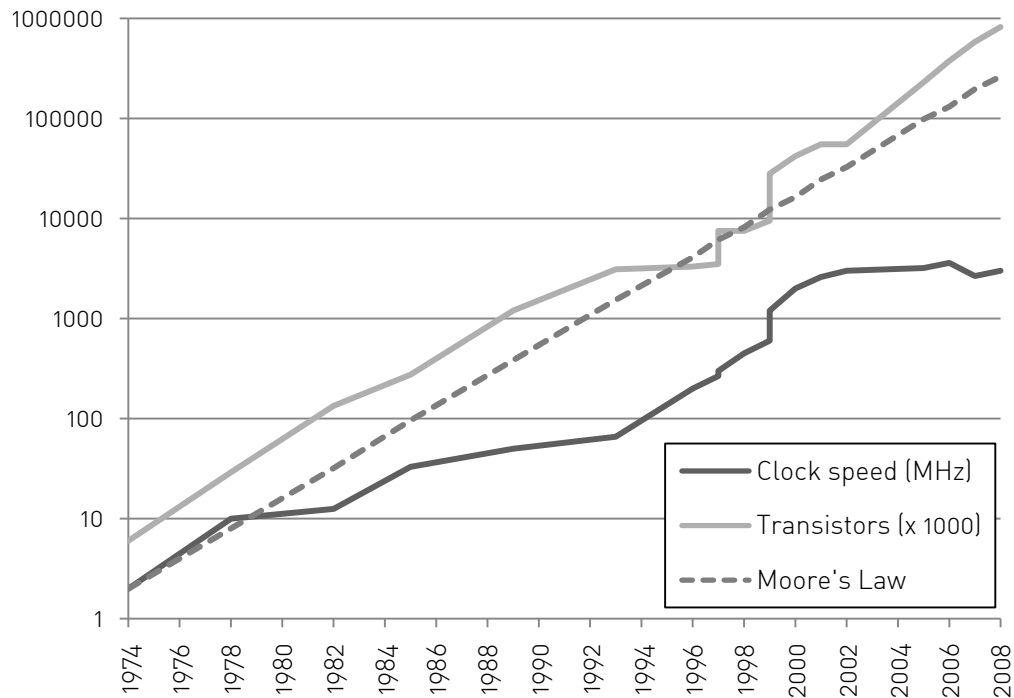


Figure 1: A comparison of the clock speeds and transistor counts of Intel CPUs³ (Sources: Intel, Wikipedia and [2]).

The transition to multi-core CPUs has some strong implications, many software developers may not yet be aware of. In the past, applications would automatically become faster as CPU clock speeds increased, but the same does not hold true for an increase in the number of CPU cores. In fact, unless an application has been specifically designed to run on a multi-core CPU, it will not benefit from the additional processing power at all. This simple fact turns a seemingly small change in hardware architecture into what some people call a revolution.

1.2. Exploiting the power of multi-core

In order to exploit multi-core CPUs, we need to write parallel applications, i.e. applications that use multiple processes or threads to perform their computations. Today, most desktop applications are single-threaded and for good reason. As I will explain in chapter 2, writing correct multi-threaded applications is diffi-

³ This is an independent recreation of a similar chart that can be found in the article "The Free Lunch Is Over" by Herb Sutter [1].

cult and when targeting single-core CPUs, it is generally not worth the additional effort. The few applications that actually use multiple threads or processes typically do so, not for performance reasons, but to hide IO or network latency from the user. On the server side, the number of multi-threaded applications is far greater. Web servers for example will often spawn many threads to process several requests simultaneously. Since at any given time, most of these threads will be waiting on IO operations, this form of multi-threading may actually improve performance even on a single-core machine. On a multi-core machine, the performance increase should be even greater.

Web servers belong to a class of application that is almost “naturally parallel”. Since the HTTP protocol is stateless, each request arriving at the web server can be handled independently. Of course multiple requests may be targeting the same resource (e.g. the same web page), but unless some form of dynamism is involved, HTTP requests do not generally “mutate” the server state. Unfortunately, not all applications have as much potential for parallelization as a web server and even if there is potential, exploiting it can often be difficult. Most database servers for example are heavily parallel, but because different database users may try to mutate the same tables or even the same table rows at the same time, measures have to be taken to ensure that the database is always left in a consistent state. Database servers typically solve this problem using transactions and locks of varying granularity.

Parallelism is commonplace on the server side, because of the scalability requirements of server applications. Developers of server applications have written parallel code for a long time and the problem field is generally well understood. An application that has to serve many concurrent users can easily be parallelized by spawning a new thread or process for each individual user. While this may not scale indefinitely, it should always be an improvement over a single-threaded solution, provided that the underlying hardware is parallel. In that sense, finding parallelism in a server application is almost trivial, but the same may not be true for a client application.

On the client, there typically is one user interacting with one application. Therefore, parallelizing a client application is very different from parallelizing a server application. If a client application is to exploit multiple CPU cores, it has to actually parallelize its computations. This may be relatively easy for some types of application (e.g. most applications that manipulate images) and very difficult for others. How would one parallelize a word processor for example? While there is some obvious potential for parallelization in a word processor (e.g. the spell checker can run in a separate thread), it clearly is not enough to keep four CPU cores busy. This very quickly leads to the question, if there even is enough parallelism in most client applications. This question will be addressed in greater detail in chapter 4.

1.3. Languages for parallel programming

Most mainstream programming languages provide some form of support for parallel programming. In the case of Java and C#⁴, it is in the form of threads and locks. While higher level abstractions are available as part of the class libraries of these languages, it is very common for developers to write against the low-level abstractions directly⁵. Considering the “high-level nature” of these languages and their general focus on safety, it may seem surprising that no higher level abstractions for concurrency were built into the languages themselves. This is especially striking, because languages like Ada⁶ and even Simula⁷ which are substantially older than Java or C# provide such abstractions as part of their syntax.

With the arrival of the first multi-core CPUs and the resulting need for developers to write parallel applications, the before mentioned mainstream languages seem increasingly ill-equipped for the task. But it is not so much the languages themselves, but their take on concurrent programming altogether that is called into question. An increasing number of developers feel that the shared-state concurrency model is inherently flawed and better alternatives need to be found. This has fostered a new interest in programming languages like Erlang⁸, a language that has a very different take on concurrency. Rather than protecting access to mutable shared state with locks, Erlang promotes the concept of “share nothing concurrency” where mutable shared state simply does not exist. Instead, the actors in a system communicate via asynchronous message passing and each message contains an actual copy of all its data instead of just a reference. This eliminates many of the problems typically associated with shared-state concurrency, but it also requires a programming style that is very different from what we are used to, today.

While the Erlang model of concurrency can be applied to functional and imperative languages alike, it is generally accepted that purely functional languages lend themselves more naturally to parallel programming than imperative languages. Especially when it comes to automatic parallelization or parallelization through annotation, most research has been done in the area of functional programming languages. This may very well be the reason, why functional languages like Haskell now enjoy greater popularity than ever before. But again, functional programming is very different from imperative programming, and at this point it

⁴ C++ which is another mainstream programming language supports concurrency only through third-party libraries. However, concurrency support is being built into the next version of the C++ standard, which is expected to be released in 2009.

⁵ A search for the keyword “synchronized” on Google Code Search yields over 400,000 results (as a comparison: searching for the keyword “finally” returns only 300,000 results).

⁶ Ada supports both shared-memory concurrency and message-passing concurrency. Its concurrency model is mostly derived from CSP.

⁷ Simula natively supports co-routines. They are however not actually executed in parallel.

⁸ Erlang is a functional, concurrent programming language developed at Ericsson.

seems unlikely, that a substantial number of programmers will abandon imperative programming altogether. It is however not surprising, that programming languages like Python or Ruby and even JavaScript, all of which heavily borrow from functional languages, enjoy great popularity. And even traditional imperative languages like C# or C++ are now slowly adopting concepts from the functional world, thereby introducing them to the mainstream.

Abandoning shared state and transitioning towards a more declarative programming model clearly has some benefits, but many developers remain skeptical. They fear that such a paradigm shift could render much of their “craftsmanship” and their experience obsolete and force them to start all over again. But this issue not only affects individual developers, it also has a major impact on businesses. Should a transition towards functional languages indeed happen, then most of the tools and processes developed at these companies would have to be discarded. Legacy code would have to be rewritten, developers would have to be re-trained and hundreds of man-years of experience would be lost. It is for those reasons that adoption of new technologies is often slow. Object-oriented programming for example was conceived in the 1960’s but has not really taken off until the 1990’s. The multi-core revolution however is based on physical limitations and its effects are not easily postponed. The slow, gradual change towards parallel programming that many people hope for is therefore unlikely to happen. But although change is necessary, it may not have to be quite as radical as abandoning an entire programming paradigm, especially one that has served us well for so many years.

One of the less radical approaches to parallel programming is software transactional memory (STM). It is often regarded as the sheet anchor for shared-state concurrency and in some way even for imperative programming. The idea behind transactional memory is very similar to that of a database transaction. Memory access is done indirectly through a transaction log. In that log, all memory reads and writes are recorded. Once a transaction is complete, the runtime will check whether any of the memory locations touched during the transaction have been modified by another thread and it will then continue to either commit the transaction or it will perform a rollback and try again. An STM implementation may use locking internally, but since locking is “optimistic”⁹, applications written using STM do not generally suffer from deadlocks. This is a major improvement over explicit locking and it reduces the complexity of shared-state concurrency to something manageable.

⁹ Locking in STM is “optimistic”, because it is assumed that threads will typically not interfere with each other and therefore no locks are taken up front. Locks are only taken to validate that memory accessed during a transaction is still in its original state before the transaction is committed.

There is little doubt that the programming languages that will prevail in the long run will be those that provide the best abstractions for parallel programming. However, what those abstractions are going to be remains to be seen.

1.4. The human factor

If we look at how much parallelism there is in our lives and how well we can deal with it, it may seem surprising that parallel programming is not second nature to us. As we navigate through life, we are bombarded with stimuli that we have to process and react to. Our eyes, ears, our nose, as well as the nerve endings under our skin constantly report to our brain their “view” of the world. We do not drown in this ocean of information, because our awareness of our surroundings is selective [2], i.e. we can focus our attention on a limited aspect of what we experience, effectively fading out everything else. Our brain still processes all the incoming information, but it may never enter our consciousness. Just how selective our attention is, can be best expressed in numbers: we take in about 11,000,000 pieces of information per second, but we can only consciously process 40 of them [3]. What that means is that while our brain is very good at parallel processing, our conscious mind is not. We can change the focus of our attention very quickly, but we cannot focus on two things at the same time.

A good example of this phenomenon is playing the piano. Playing the piano requires a fair amount of parallelism. A piano player needs to read music and play different notes with each hand simultaneously. Of course this seems to contradict the earlier statement that we cannot focus on two things at the same, but what is important to note is that a piano player will do most of these things subconsciously. A novice piano player on the other hand will have to devote much of his attention to translating the individual notes into movements of the hand. This will force him or her to “context-switch” between reading music and playing the notes resulting in pauses or slowdowns.

If we apply the same reasoning to programming, then parallel programming should no longer be difficult for us, once we are skilled enough to reason about our code subconsciously. Unfortunately however, writing software for the most part is a conscious process and does not generally follow a scheme that is simple enough to completely internalize. For that reason, writing software while considering all the complex interactions between different threads may very well be beyond what our brain is capable of. This does not however mean that we are not at all capable of writing parallel software. It only means that we need to make sure that we design our parallel programs in such a fashion that we can reason about each process individually.

2. Concurrent programming today

In this chapter, we will examine the shared-state concurrency model that is the basis for most of today's concurrent applications. We will try to explain the popularity of this model and examine why many programmers are now looking for alternatives.

2.1. Threads and shared state

Most concurrent programming today is done using the shared-state concurrency model. There are two main reasons for this: primarily, it is because the APIs exposed by operating systems tend to follow that model, but also because it coincides well with the imperative programming paradigm. In a system that follows the shared-state concurrency model, actors (called "threads") communicate with each other by mutating state which they share. Unlike in a more process-oriented model where actors can only communicate through specific message channels, in a shared-state system, all actors reside in the same address space. This has the obvious advantage that communication between actors can be very cheap. For this reason, threads are often referred to as "lightweight processes" as opposed to "heavyweight processes" which all have their own address space and which can only communicate by taking the expensive route through the OS kernel.

The notion of mutating state is the foundation of the imperative programming paradigm and while efforts have been made to reduce the scope of that state by encapsulating it into classes, the basic premise is still the same. In that sense, the shared-state concurrency model appears to be a good fit for imperative programming languages, but as I will discuss on the next few pages, it may very well not be.

2.2. Race conditions

By allowing multiple threads to access the same data, a new problem arises. Because concurrency is by definition non-deterministic, there is no guarantee that threads will access the shared data in an order that will yield a desired result. In fact, in a truly parallel environment, not only can threads access shared data in any order, they can access the data at exactly the same time. This of course leads to completely unpredictable results. To demonstrate this effect, I will now give a

simple example¹⁰: in a banking application, we would like to be able to transfer a specified amount of money from one account to another. This transaction is composed of two separate operations, one of which is “withdraw” where we take money out of an account and “deposit” where money is added to another account. Combining these two operations into a new “transfer” operation is now almost trivial, as shown in Figure 2.

```
procedure Transfer(from, to, amount)
  result := call Withdraw(from, amount)
  if result = true then
    call Deposit(to, amount)
  end if
```

Figure 2: The Transfer procedure

However, while this operation may work just fine in a sequential application, it is not guaranteed to work in a concurrent application. Because the “transfer” operation is not atomic, i.e. it does not consist of a single indivisible operation, there is a point during its execution, where the state of the system is inconsistent. That is when money has been withdrawn from one account, but it has not yet been deposited to the other. Should another thread execute a similar operation before the consistency of the system is restored, it will do so under false presumptions. This situation becomes even worse if we assume that neither the “withdraw” nor the “deposit” operations are atomic. The “withdraw” operation might be implemented as shown in Figure 3.

```
procedure Withdraw(account, amount)
  balance := call GetBalance(account)
  if balance < amount then
    return false
  else
    newBalance := balance - amount
    call SetBalance(account, newBalance)
    return true
  end if
```

Figure 3: The Withdraw procedure

The problem with this implementation is that the actual withdrawal is done in three separate steps. First, the current account balance is retrieved, then the specified amount is subtracted and finally the new balance is stored. During any of these steps, the state of the system is inconsistent and another thread may set

¹⁰ The money transfer example is found in many books and papers on concurrency and it is therefore difficult to determine, where it originated. The author of this thesis however does not take any credit for it.

a new account balance while the first thread continues to work with the value it initially retrieved and which is now “stale”. Once the first thread completes its computations and calls “SetBalance” to update the account, the update made by the second thread is lost¹¹. In other words, the outcome of the operation depends on the way the two threads interleave which in turn is completely non-deterministic. Figure 4 illustrates this situation.

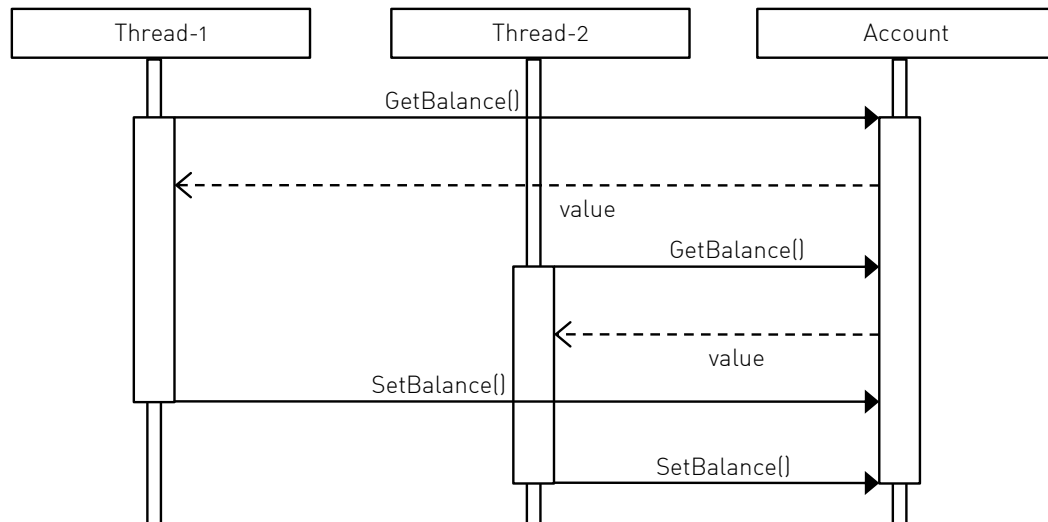


Figure 4: A sequence diagram illustrating the lost update problem

The particular problem with race conditions is that they occur randomly, meaning they can occur once and then never again, they can never occur on one particular machine but occur frequently on another, or they can show up years after the initial release of a product in a particular configuration that has not been tested for. Worse still, the effects of a race condition may go unnoticed for a long time. Similar to a buffer overflow for example, a race condition will often not crash the affected application. Instead, it will lead to undefined behavior that can either be quite noticeable or have no visible effects at all. In one particularly noteworthy incident, a medical radio-therapy machine called the Therac-25 subjected six patients to a lethal dose of radiation. One of the main causes of this malfunction was later found to be a race condition in the control software that would only occur, if the operator of the machine would change the setup too quickly [3]. While this is obviously a rather extreme example, it shows that race conditions are not to be taken lightly.

2.3. Locks

To prevent race conditions from happening, a mechanism is needed to serialize access to shared data. Operating systems and also many programming languages

¹¹ This is commonly referred to as the „lost update problem“ in the database world.

provide such a mechanism in the form of locks. Using locks, a programmer can ensure that operations which are composed of several sub-operations appear to other threads, as if they were executed atomically. This means that (ideally), no thread ever gets to see the world in an inconsistent state.

If we were to apply locking to our “Transfer” example from the previous section, we might get something similar to what is illustrated in Figure 5.

```
procedure Transfer(from, to, amount)
  call Lock(transfer_lock)
  result := call Withdraw(from, amount)
  if result = true then
    call Deposit(to, amount)
  end if
  call Unlock(transfer_lock)
```

Figure 5: Locking applied to the Transfer example

On entering the “Transfer” operation, we acquire a global lock that we release, only after the entire transaction is complete. Another thread attempting to perform the same operation at the same time would also attempt to acquire the lock, but as it would already be taken, the thread would have to block until the lock becomes available again. This form of locking where the lock can only be held by one thread at a time is typically called a “mutex” which is a contraction of the words “mutual exclusion”.

2.3.1. Mutual exclusion

The mutual exclusion mechanism used in Figure 5 is implemented as two functions “Lock” and “Unlock”. While many programming languages provide mutual exclusion as part of their syntax, it is in fact possible to implement a mutex on the language level. A heavily simplified but functional implementation of the “Lock” and “Unlock” function is given in Figure 6.

```
procedure Lock(the_lock by reference)
  while call TestAndSet(the_lock) do
    skip
  end while

procedure Unlock(the_lock by reference)
  the_lock := false
```

Figure 6: Implementations of the Lock and Unlock functions

In this example, we assume that the lock variable is of type “boolean” and that it is passed to the “Lock” and “Unlock” functions using the call-by-reference convention. The “Lock” function consists of a single while-loop that attempts to acquire

the lock using an atomic test-and-set operation. The purpose of this operation is to conditionally and atomically write a new value to the lock variable. If the value of the lock variable is “false”, i.e. if the lock is not currently being held by another thread, the variable is set to true and the function returns false. Otherwise the test-and-set operation returns true and the “Lock” function will continue to wait on the lock until it becomes available again. Since the test-and-set operation needs to be atomic on the instruction level, it has to be implemented in hardware. However, for better understanding an exemplary implementation of test-and-set is given in Figure 7.

```
procedure TestAndSet(the_lock by reference)  
  atomic  
    current := the_lock  
    the_lock := true  
    return current  
  end atomic
```

Figure 7: An exemplary implementation of test-and-set

While the mutex-implementation given in Figure 6 may work as expected, it has the undesirable property that it busy-waits on the lock, meaning it heavily strains the CPU without making any real progress. This is particularly adverse on a time-sharing system where the thread that is holding the lock cannot make any progress while the thread waiting on the lock is occupying the CPU. To avoid this effect, mutexes are typically running on the kernel, allowing the operating system to assign smaller time slices to threads that are waiting on locks. This however requires an expensive context-switch, which is why some mutex implementations will busy-wait in user mode for a while and switch to a kernel mode lock only after some amount of time has passed.

2.3.2. Locking granularity

In Figure 5, we protected our transfer example against race conditions using a single global lock. The advantage of this approach is that it makes the code very easy to reason about. Since the lock guarantees that the Transfer function is only ever entered by one thread at a time, we can now reason about our program as if it was sequential. The problem is that our program not only appears to be sequential, it essentially has become sequential. While we can spawn any number of threads that all attempt to perform a transfer operation, only one of those threads will actually make progress at any given moment. Also, because waiting on a lock is an expensive operation, the parallel version of the transfer example is likely to perform worse than the sequential version.

Sequential bottlenecks like the one just described are common in concurrent applications, because coarse grained locks are much easier to reason about than

fine grained locks (see 1.4). If however we are to actually benefit from parallelism, we need to strive for the finest locking granularity possible.

In the current version of our transfer example, our use of locking focuses on code rather than on data. Two threads operating on two completely disjoint sets of accounts could safely do so in parallel, but our current implementation of the transfer function forbids that. We could therefore improve our code by taking one lock per account instead of one lock for all accounts. Since neither the “Withdraw” nor the “Deposit” functions are currently thread-safe, this seems like a good place to start. An updated version of the “Withdraw” function is given in Figure 8.

```
procedure Withdraw(account by reference, amount)
  call Lock(account)
  balance := call GetBalance(account)
  if balance < amount then
    return false
  else
    newBalance := balance - amount
    call SetBalance(account, newBalance)
    return true
  end if
  call Unlock(account)
```

Figure 8: The thread-safe Withdraw function

The updated function now takes a lock at the beginning and releases it after completing the transaction. Also, instead of using a global lock, we now pass the account variable to our “Lock” and “Unlock” functions. This obviously would not work with our current implementations of these functions as they assume that the lock variable is of type “boolean”. However, the adjustments needed to make this work are rather simple¹² and not subject of this thesis.

The modified “Withdraw” function is now thread-safe and as it uses one lock per account, two threads making a withdrawal from two separate accounts can now run in parallel and without interfering with each other. Similarly, we can update the “Deposit” function so that it possesses the same qualities.

2.3.3. Composability

Until now, we have looked at the “Withdraw” and “Deposit” functions individually. Our initial goal however was to write an improved version of the “Transfer” operation, which as illustrated in Figure 2 is a composition of these two functions. Un-

¹² Provided that the account variable is of a structural type, a hidden field “lock” of type boolean could easily be appended automatically by the runtime. This would allow us to reuse our original implementations.

fortunately, while “Withdraw” and “Deposit” are thread-safe on their own, their composition is not. In order to ensure that these two functions can execute as if they were a single atomic operation, we have to take all the locks used by both of these functions up front. The only way of doing that however, is to break the abstraction and actually look inside the implementation of these functions to determine, which locks need to be taken. Of course this heavily contradicts the object-oriented programming model and makes it very difficult for developers to interact with lock-oriented code written by third parties.

In our particular example, we need to obtain exclusive access to both the source account and the destination account before calling “Withdraw” and “Deposit”. An updated implementation¹³ of the “Transfer” function is given in Figure 9.

```
procedure Transfer(from by reference, to by reference, amount)
  call Lock(from)
  call Lock(to)
  result := call Withdraw(from, amount)
  if result = true then
    call Deposit(to, amount)
  end if
  call Unlock(from)
  call Unlock(to)
```

Figure 9: Taking the two locks in the Transfer function

Because we now only lock the accounts that we are operating on, multiple threads can perform transfers between unrelated accounts in parallel. Unfortunately, while the code given in Figure 9 may not look like it is incorrect, it is prone to a common locking-related problem known as a “deadlock”.

2.3.4. Deadlocks

A deadlock is a circular dependency between two or more threads, where each thread waits for another thread to release a particular lock that it itself wishes to acquire. In the presence of a deadlock, a program is no longer able to terminate by itself and therefore has to be aborted from the outside. Similar situations can occur in real life, for example in traffic at an intersection where the right-of-way is not immediately apparent. This however is far as the analogy goes. In such situations, drivers typically signal each other to reconcile who gets to go first. In a multi-threaded system however, this option does not generally exist.

¹³ For this example to actually work, the “Lock” and “Unlock” functions would have to be modified so that a call to “Lock” would not wait on a lock that is already held by the calling thread.

Deadlocks occur, when locks are taken in an incorrect order. This is best demonstrated with an example (see Figure 10).

```
spawn14 Transfer(account1, account2, 100)
spawn Transfer(account2, account1, 100)
```

Figure 10: A typical deadlock situation

If we assume, that each call to the “Transfer” function is made by a different thread and that the threads interleave in a particular way, the situation depicted in Figure 11 may arise.

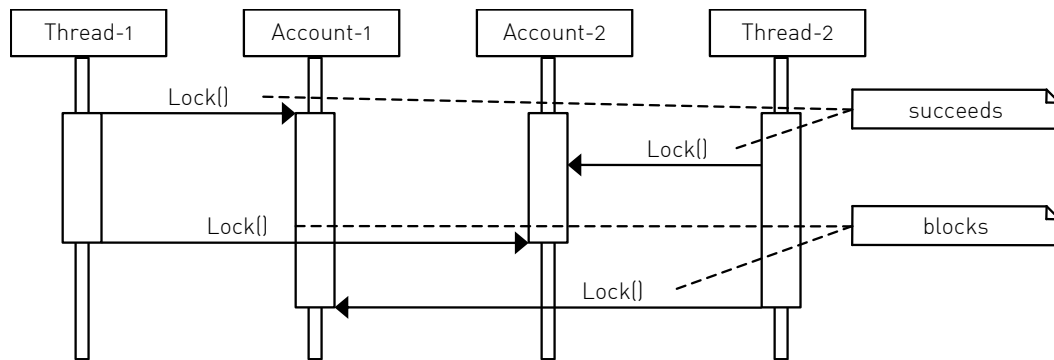


Figure 11: Sequence diagram of a typical deadlock

In prose this means that once each thread has acquired its first lock, it fails to acquire the second lock and subsequently enters a wait-state. Because now both threads are waiting, neither one can release the lock it is already holding, which in turn means that neither thread can leave their wait-state.

Before we look at how this problem can be resolved, it is important to know that the potential for deadlocks usually is not as apparent as it is in this particular example. Deadlocks often occur when a function that is holding a lock calls another function that unknowingly attempts to acquire the same lock. As a rule of thumb, it is therefore advisable not to call functions written by third-parties while holding a lock. This again demonstrates how locking is inherently non-composable and how building a modular system with a locking-oriented concurrency model can be extremely difficult.

As mentioned earlier, when acquiring multiple locks, the order in which they are acquired is very important. However, it does not suffice to use a consistent ordering in one particular function. If we modified our “Transfer” function so that it would always use the same locking order for two given accounts, a deadlock may still occur when another function acquires the same locks in a different order.

¹⁴ The “spawn” instruction is introduced here as a means of calling a function in an asynchronous fashion.

This can be prevented by defining an ordering relation for locks. Assuming that a lock is implemented as an object, the object-id or its memory address are suitable candidates for such a relation. With an ordering relation in place, we can change the lock acquisition code in our “Transfer” function to the following (see Figure 12).

```
if from < to then
  call Lock(from)
  call Lock(to)
else
  call Lock(to)
  call Lock(from)
end if
```

Figure 12: Ordered lock acquisition

Provided that all other functions in our application use the same mechanism to determine the correct locking order, a deadlock is no longer likely to occur. However, since this form of deadlock prevention relies solely on the programmer’s caution and is not commonly enforced by the compiler or the runtime, no guarantee can be given that the correct locking order is obeyed all throughout an application. This is especially problematic because deadlocks, like race conditions, are non-deterministic and they may not show up during development or testing.

2.4. Correctness

Many programmers try to avoid concurrency whenever they can, fearing that it may introduce additional complexity and a whole new class of concurrency-related errors. This fear is not at all irrational and while there is little empirical data to support the claim that concurrent programs are more prone to error than sequential programs, anyone who has written concurrent programs using threads and locks will agree that this is usually the case. Until now, this has not been especially problematic since most concurrent programming was done in the field of systems programming and in the server and mainframe environment, where the actual implementation of concurrent algorithms was typically executed by concurrency experts. As we enter the multi-core era however, more and more application developers will have to write concurrent code and they may not have the same level of expertise. As a result and unless we can find and adopt safer concurrency models, it may very well be that the reliability and stability of application software will go down dramatically in the future.

Once we introduce concurrency to our applications and thus non-determinism, most techniques we have developed during the last few decades to validate the correctness of our code will no longer work as expected. Unit tests for example have proven themselves useful for exposing all kinds of bugs in sequential pro-

grams, but because concurrency-related bugs are non-deterministic, so is the outcome of a unit test for a concurrent program (or module for that matter). That is not to say that unit tests are generally not applicable to concurrent programs. An assertion that checks an invariant for example can help in exposing a race condition. But since a race condition occurs at random, an assertion is in no way guaranteed to detect such an error. Similarly, static analysis tools can help in detecting certain types of concurrency related problems, but they cannot provide any certainty that a concurrent program will behave correctly at runtime.

The difficulty with concurrent programs is that the absence of noticeable errors has no significance for the correctness of a program whatsoever. Merely testing for race-conditions and deadlocks is therefore not enough. We would much rather like to know with definitive certainty, that our programs are correct and that under no circumstances are race-condition or deadlocks going to happen at runtime. Such proofs of correctness are not uncommon in highly critical fields such as avionics or in military applications, but they require a much more confined concurrency model where processes communicate over well-defined channels rather than by mutating shared state. One concurrency model that provides the kind of formalisms required for proving the correctness of concurrent systems is that of "Communicating Sequential Processes" (CSP) [4]. It has been a major influence to many programming languages, especially Occam¹⁵, Ada and Erlang but until now, it has largely been ignored by mainstream languages.

¹⁵ Not to be confused with OCaml. Occam is an imperative programming language that implements many of the ideas of the CSP algebra.

3. Alternative approaches to concurrency

In this chapter, we will look at two alternative models for concurrent programming: software transactional memory which builds on top of the shared-state concurrency model and message passing concurrency, where no state is shared. We will discuss the advantages and disadvantages of each model and examine, which of the challenges that concurrent programming poses are addressed by each model.

3.1. Software transactional memory

In 2002, Steve Gilheany, a systems engineer from Manhattan Beach, California created a chart that predicts that by 2007 we would have CPUs running at 24 GHz [7]. This prediction may seem ludicrous now, but it was uttered at a time where clock speeds were still on the rise and an end to this trend was not in sight. When three years later, Herb Sutter wrote his article “The Free Lunch Is Over” [1], it caused quite a stir. Interestingly, the article did not only come as a surprise to regular developers, but to language designers and –implementers as well. This lack of premonition, if one wants to call it that, has given hardware manufacturers a considerable head start, forcing developers to follow suit, and quick. As learning new programming languages and potentially even new programming paradigms takes time, many developers now hope that the languages they are already familiar with will evolve to better support this new need for concurrency. The demand for better concurrency abstractions may also have sparked a new “foot race” between language designers and whichever language provides the best abstractions the soonest, may very well draw to it a large group of new followers.

One of those new abstractions is software transactional memory, and it may not come as a surprise to learn that both Sun Microsystems and Microsoft are heavily betting on it.

3.1.1. Shared-state concurrency revisited

In chapter 2, we looked at the difficulties of concurrent programming using the shared-state concurrency model. We determined that the two main problems with this model are composability and correctness. We also established that the primary reason for why it is so difficult to write error-free programs using the

shared-state concurrency model, is the fact that it relies on the programmer rather than the compiler or the runtime to ensure that shared data is protected and that locks are taken in the right order. When we say shared-state concurrency however, what we really mean is the concurrency model of threads and locks.

The idea of software transactional memory is similar to that of threads and locks because it is also built on top of the shared-state concurrency model. Unlike threads and locks however, software transactional memory does not suffer from composability issues and because it is much easier to reason about, it is also much harder to get wrong.

3.1.2. Memory transactions

Transactions are a familiar concept in the database world and while memory transactions serve a slightly different purpose, the semantics are essentially the same. Database transactions are required to be atomic, consistent, isolated and durable¹⁶. Atomic refers to the fact that a transaction happens either in its entirety or not at all, consistent means that the state of the database is consistent before and after a transaction, isolation ensures that different transactions do not interfere with each other and durable means that after a transaction is complete, the changes it made to the database are persistent. With the exception of durability¹⁷, a memory transaction possesses all of these qualities.

What is just as important as the properties of a transaction however, is the fact that they are guaranteed by the database system and that the client is completely left out of the equation. The same applies to an STM system, where the developer never has to take locks explicitly. This is the most important advantage of STM over the threads-and-locks concurrency model and it explains why so many developers have such high hopes for this technology.

In a programming language that supports software transactional memory, developers annotate code that is not thread-safe so that the runtime can execute it as part of a transaction. This concept is familiar to developers who have already been exposed to concurrent programming, as the same passages of code would otherwise have to be protected by a lock. Developers who have not yet done any concurrent programming on the hand will have to learn about thread-safety but not about the intricate details of locking.

To demonstrate how relatively easy it is, to ensure thread-safety using memory transactions, we will now look at a modified version of the money transfer example from chapter 2. Figure 13 illustrates how the code would have to be changed, to use transactional memory.

¹⁶ The sum of these properties is usually referred to as „ACID semantics“

¹⁷ That is because memory is non-persistent and will not for example survive a reboot

```
procedure Transfer(from, to, amount)
  atomic
    result := call Withdraw(from, amount)
    if result = true then
      call Deposit(to, amount)
    end if
  end atomic
```

Figure 13: The transfer function using software transactional memory

In 2.3.2 we established that coarse-grain locking, while often responsible for poor performance, is much easier to reason about than fine-grain locking. If we used a single global lock for our entire application, we would essentially eliminate all parallelism, but at the same time locking would become a non-issue. In a perfect world, we would therefore like to write our applications as if they used a single global lock, but still have the level of parallelism we would get from locking each resource individually. Software transactional memory comes very close to this ideal. To make our transfer function thread-safe using STM, all we have to do is to enclose the code in an “atomic”-block. This small change alone gives us the guarantee, that the code within that block is executed atomically and that no other thread tampering with the same resources will see them in an intermediate state. At the same time, we can confidently call other functions without having to worry about deadlocks. This radically changes the way we think about shared-state concurrency and it drastically reduces the possibility of error.

3.1.3. Inside software transactional memory

As with all good abstractions, the interfaces provided by software transactional memory are simple. The underlying implementation of an STM system however can be extremely complex. While a full understanding of that complexity is not required to use STM to one’s advantage, understanding at least the basic concepts behind an STM implementation can help in avoiding some of its pitfalls.

A transaction consists of two separate phases. In the first phase called the “recording-” or the “speculative execution phase”, the code that makes up the transaction is executed and all its memory reads and writes are recorded in a log. The entirety of all memory locations read by a transaction is called its “read set” while the locations written to are called its “write set”. Because a transaction needs to be without effects in the case that it should have to be rolled back, any memory writes performed during a transaction can only be done virtually, i.e. the original values cannot be overwritten. After the code for the transaction is executed, the second phase called the “commit phase” is initiated. During this phase, the STM runtime validates that the transaction’s read set has not been touched by another thread while the transaction was executing. If validation fails, the transaction is rolled back, meaning the transaction log is discarded and the code inside the

transaction is executed again from the beginning. If on the other hand the read set is still in its original state, the transaction is committed by repeating the previously recorded memory writes at their target locations.

To better illustrate this principle, an example of a failed transaction is given in Figure 14. In this example, two transactions operate on the same bank account. The first transaction reads the account balance as well as the account's interest rate from memory, then multiplies the balance by the interest rate and writes the new balance back to memory. At the same time, a second transaction retrieves the interest rate from the same account, increases it by 0.5 and then writes that value back to memory as well. Since the second transaction finishes before the first, it commits its changes, thereby causing the first transaction's read set to become invalid. Once the first transaction completes, it checks its read set, and as it is no longer valid, it aborts and tries again.

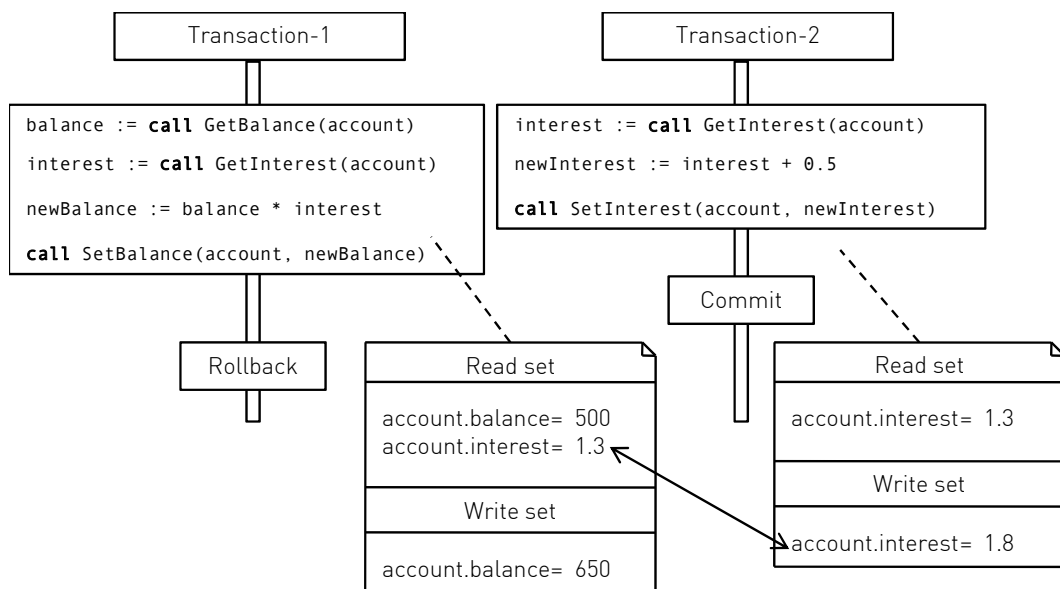


Figure 14: A memory transaction dissected

Put differently, a transaction fails when there is an intersection between its read set and the write set of another transaction that commits before it. STM implementations can use this fact to validate a transaction's read set very quickly and without having to actually compare memory contents. This however means that transactions are only atomic with respect to other transactions. When shared-data is accessed outside of transaction, race conditions may still occur.

The concepts we have discussed so far are the basic building blocks of any STM implementation. We have however yet to explain how a transaction is actually committed. This is where different STM implementations take different approaches. The primary problem with committing a transaction is that both the validation of the read set and the execution of the memory writes have to happen atomically. Otherwise, a second transaction could modify the read set of the first transaction

after it has been validated but before the memory writes have been executed. In a concurrent system, atomicity can be achieved through locking, and there are in fact some STM implementations that use traditional locking internally. More commonly however, STM is implemented lock-free. What this means is that at least one thread in an STM system will always make progress or in other words, a lock-free STM implementation cannot deadlock. Lock-free algorithms are implemented using atomic primitives provided by the hardware, most notably the compare-and-swap (CAS) operation. This operation is similar to test-and-set (see Figure 7), but it operates on arbitrary values and performs an actual comparison. An exemplary implementation of compare-and-swap is given in Figure 15.

```
procedure CompareAndSwap(value by reference, expected, new)
  atomic
    current := reference
    if current = expected then
      reference := new
    end if
    return current
  end atomic
```

Figure 15: An exemplary implementation of compare-and-swap

Compare-and-swap typically operates on a single machine word. Additionally, some instruction sets support a “double-compare-and-swap” (DCAS) instruction¹⁸ that operates on two machine words at the same time. For smaller read-sets, lock-free STM implementations can use CAS or DCAS to validate and commit a transaction with a single instruction. As read sets grow larger however, STM systems often have to fall back to using complex algorithms to ensure the atomicity of a commit. Many STM implementers therefore hope for better hardware support in the future¹⁹ as this could help improve performance dramatically.

3.1.4. Condition variables

A common way of synchronizing threads is to use condition variables. Using condition variables, a thread that expects a particular input from another thread can block until that input is available. Because the operating system can be made aware of the condition variable, the scheduler can skip waiting threads for as long as the condition variable is not in its signaled state. This allows other threads to run without interruption.

¹⁸ DCAS was supported by some Motorola CPUs and is going to be supported by Sun’s upcoming “Rock” CPU (see next footnote). The x86 instruction set provides a CMPXCHG8B instruction [9] that is very similar to DCAS in that it operates on two machine words. However, the two-word value has to be in contiguous memory.

¹⁹ Sun Microsystems is currently developing a SPARC-family processor codenamed “Rock” that natively supports transactional memory [8].

Condition variables are a form of locking and as such do not mix well with software transactional memory. However, since they are an important thread coordination mechanism, they are often seen as being indispensable. Fortunately, STM lends itself to a very similar mechanism that can be implemented almost entirely using constructs that STM already provides. To better illustrate this mechanism, we will now look at a modified version of the “withdraw” function from chapter 2 (see Figure 16).

```
procedure Withdraw(account, amount)
  atomic
    balance := call GetBalance(account)
    if balance < amount then
      retry
    end if
    newBalance := balance - amount
    call SetBalance(account, newBalance)
  end atomic
```

Figure 16: A blocking version of the Withdraw function

In this version, the “withdraw” function no longer returns false when the account balance is lower than the amount that needs to be withdrawn. Instead, it uses the new keyword “retry”²⁰ to instruct the STM implementation to abort the transaction and try again. In other words, the modified code now blocks until the account balance is sufficiently high.

If “retry” simply performed a roll-back like the STM implementation would, if the transaction had failed, it would essentially busy-wait on the account balance and consume a lot of CPU resources in the process. However, the STM implementation knows the transaction’s read set and it also knows that the only way the transaction will not run into the “retry” again is if its read set changes. Furthermore, it knows that the only way the transaction’s read set is going to change is if it intersects with the write set of another transaction. Therefore, whenever a transaction invokes a “retry”, the STM implementation will put that thread to sleep, remember the transaction’s read set, compare it to all successful commits made by other transactions and wake up the retrying transaction, the moment its read set is changed [8]. This mechanism makes for a useful and safe replacement for condition variables and it goes to show, how the additional information available to an STM system can be used in interesting ways.

²⁰ The retry keyword is specific to the “Concurrent Haskell” implementation of software transactional memory, although other implementations may have adopted the concept as well.

3.1.5. Exception safety

Exception safety plays an important role when working with explicit locking. When an exception is raised in a function that is holding a lock, measures must be taken to ensure that the lock is released, before the stack is unwound. Otherwise, the lock will be held indefinitely and a deadlock will occur. Unfortunately, the release of a lock in the event of an exception can itself be the cause of a problem. If a lock is released before the computations it protects are complete, other threads may be exposed to the intermediate results of that computation which in turn can lead to race conditions. Therefore, additional precautions have to be taken to restore the consistency of the system before releasing any locks. Of course this is extremely cumbersome and difficult to get right.

Software transactional memory on the hard supports exceptions in a much more intuitive fashion: if an exception is raised inside an “atomic” block, the transaction is rolled back and the exception is propagated to the next handler. This is safe, because a transaction does not hold any locks or modify any share data until it is committed. Alternatively, the exception can be handled inside the “atomic” block, in which case the transaction continues uninterrupted.

3.1.6. Limitations

Software transactional memory may be much more convenient than explicit locking, but because it is an abstraction it is also less universally applicable. Most notably, code that is part of an STM transaction cannot have side-effects on non-memory resources. This becomes quite self-evident, when one recalls that a transaction may have to be aborted and retried any number of times. Because a transaction log only records memory writes, any changes made to the file system or any other non-memory resource would persist, even after the transaction has been rolled back. This could be prevented, at least in part, by buffering all output. However, the feasibility and scalability of this solution has yet to be examined. More importantly however, STM does not mix well with user input. A transaction that prompts the user to input a value would do so repeatedly, if it failed to commit the first time. In practice however, this should rarely be a limitation. Nonetheless this goes to show, that software transactional memory is not a drop-in replacement for explicit locking.

A second area where problems can arise is transaction length. STM relies on the common observation that more often than not, threads operating on the same resources never interfere with each other. For this reason, STM does not take locks up front, which generally leads to more concurrency. Unfortunately, while this approach may work well for short-running transactions, it is less effective for long-running transactions. A long transaction, meaning a transaction that takes a long time to execute, is far more likely to have its read set invalidated by another transaction. This will cause it to abort and retry, which of course is particularly

undesirable for long running or “slow” transactions. In the worst case, a transaction will end up in an abort-retry cycle because another transaction keeps committing before it. Some STM implementations will therefore temporarily assign a higher priority to threads that continuously fail to commit. This is problematic however, because it defies the control of the programmer and changes the performance properties of the application. Therefore, overly large transactions are best avoided – even more so, because they often hint at design flaws.

3.1.7. Performance

Because software transactional memory is optimistic and can use fine grain locking internally, one would expect it to be just as fast as or even faster than explicit locking. On the other hand, STM has to do a fair amount of bookkeeping, which can sometimes outweigh the performance gained from holding the locks for a shorter amount of time. Because the cost of maintaining the transaction log is proportional to the size of the transaction and the gain in concurrency is proportional to the number of threads as well as the number of available CPU cores, the actual performance benefit or -cost of transactional memory is difficult to express in numbers. Some early tests have shown that STM is on average 10-50% slower than explicit fine grain locking and can be up to 200% slower in the worst case [8]. Other experiments however, have come up with very different results [9].

The real question however is, whether these numbers actually matter. If software transactional memory helps us exploit a level of parallelism that we would otherwise be unable to exploit, then any performance penalty is easily justified; even more so, because at this time, performance reserves are vast and largely underutilized.

3.1.8. The garbage collection analogy

Managed programming languages like Java or C# enjoy great popularity, but this has not always been the case. While managed languages have been around for a long time, there has often been a performance dogma associated with them and a lot of programmers felt that these languages did not leave them enough room for micro-management. But as faster CPUs and better language implementations have narrowed the performance gap between managed- and unmanaged languages, many programmers have gladly traded some level of control for type-safety, security and composability.

In many unmanaged programming languages, memory management is very explicit. When allocating memory, it is the responsibility of the programmer to make sure that memory is freed when it is no longer needed. This may not seem like a problem at first, but when sharing memory resources between different components of a system, it may not always be clear when a resource is no longer needed. Manual memory management is therefore inherently non-composable.

Managed programming languages solve this problem by eliminating the need to manually free memory resources. Instead, these languages rely on automatic garbage collection to make sure that memory is made available for future allocations when it is no longer referenced. By also banning pointer-arithmetic, these languages effectively abandon the notion of memory altogether. As with all abstractions, this leaves the programmer with less control, but it also eliminates a whole class of errors.

The parallels between automatic memory management and software transactional memory are numerous: just like STM, garbage collection primarily solves a composability problem. Garbage collection applies only to memory resources as does software transactional memory. In early implementations, garbage collection had a substantial overhead over manual memory management, as do early STM implementations. The list goes on.

While automatic memory management was met with a lot of skepticism at first, it is now widely accepted that at least for application level programming, the advantages of garbage collection largely outweigh its disadvantages²¹. Similarly, software transactional memory might replace explicit locking in some areas, as soon as more mature implementations become available, especially because many programmers are already familiar with the concept of transactions from writing database applications.

3.1.9. Conclusion

Software transactional memory is a technology that is still in its infancy. While implementations exist for many different programming languages, most of them have yet to reach a maturity level that allows programmers to use them with confidence. Also, since many STM implementations are built on top of existing programming languages as libraries, rather than into the languages themselves, their syntax tends to be much more verbose than the “fictional” syntax used in this chapter. It may therefore be another five to ten years until software transactional memory will begin to play a significant role in software development. While Sun Microsystems and Microsoft as well as other companies have announced that they are researching STM, they have yet to reveal how this research will impact programming languages like Java, C# or C++.

Software transactional memory is often made out to be a quick remedy for the problems of shared-state concurrency. But while it may very well be that remedy,

²¹ Modern generational garbage collectors compare favorably performance wise to explicit manual memory management. However, automatic memory management is sometimes criticized for being non-deterministic which makes it less useful for real-time applications. Also, many garbage collected languages do not support RAII, making it hard to dispose of non-memory resources (e.g. file handles) when they are no longer needed.

it seems unlikely that it will arrive in time for when the demand for parallelism in client applications reaches a point where it can no longer be ignored. But even if it does arrive in time, there are some skeptics who believe that pursuing software transactional memory is “barking up the wrong tree”. They feel that the shared-state concurrency model is so deeply flawed, that not even software transactional memory can redeem it [11]. Instead, they believe that mutable shared-state needs to be abolished as a whole and communication between threads needs to happen in a much more controlled fashion. In fact, as we will discuss in chapter 4, the problem of inter-process communication is only part of the challenge of the multi-core revolution. This in turn means that software transactional memory can also only be part of the solution.

Because concurrency is such a complex problem, a single solution is unlikely to address all aspects of it. This makes it even more complex for language- and library implementers to decide, which concurrency abstractions to pursue and which to ignore. Software transactional memory shows great promise, but whether it is going to be widely adopted remains to be seen.

3.2. Message passing concurrency

In a system that uses shared-state concurrency, the communication between threads is unbounded. Any thread can mutate any shared variable that it has access to. This makes it very difficult to reason about even the sequential parts of an application, as the outcome of a computation may depend on the state of a variable that is controlled by another thread. Of course programmers have learned to tame shared-state and a skilled programmer will limit the scope and accessibility of shared-state to a minimum. But what this means is that correctness is something that requires discipline and not something the underlying infrastructure enforces. Critics of the shared-state concurrency model on the other hand argue that that correctness should be the default and that the input- and output channels provided by an actor should be well defined, so that they can be validated. This closely relates to functional programming, where functions have well defined inputs and outputs and the return value of a function depends solely on its input. But the desire for a less complex concurrency model is one that functional- and imperative programmers share.

Established languages like Java, C# or C++ have yet to find the answers for the questions raised by the multi-core revolution, forcing many programmers to look elsewhere. This has sparked new interest in alternative programming languages and at the center of which is Erlang. Erlang is a functional programming language that features strong built-in support for concurrency [12]. It follows the mathematically founded “actor model” that promotes message passing as the only means of communication between processes.

On the next few pages, we will take a closer look at “Erlang-style concurrency”, representative for a wider array of different implementations of message passing concurrency. Because Erlang is a functional language that is syntactically similar to Prolog and as such may be difficult to understand for readers more accustomed to imperative languages, I will continue to use the pseudo-code notation introduced in chapter 2. This also aids in demonstrating that while Erlang is in fact a functional language, the Erlang concurrency model can be applied to imperative languages as well.

3.2.1. Actors and messages

The basic building block of a concurrent Erlang application is the actor. An actor is an entity that can receive messages and upon receiving these messages, can perform specific operations. Additionally, an actor can itself spawn new actors or it can send messages to actors that it is familiar with. Each actor has a unique identifier (sometimes called a process ID), that functions like an address, allowing other actors to send messages to it. From the outside, an actor appears as a black box with a single entrance for incoming messages and a single exit for outgoing messages. This means that a change in the actor can only be triggered by an incoming message and not by an arbitrary event in the environment. In other words, a specific sequence of messages sent to an actor will always yield the same results, regardless of the state of the rest of the system. This makes reasoning about an individual actor very simple. Additionally, because an actor will only process one message at a time, all the code that makes up an actor is entirely sequential. Because an actor can still receive messages while it is busy processing a previous message, it has to maintain a queue that all new messages are added to. In Erlang terminology, this queue is called a “mailbox”.

Because Erlang is a functional language, actors are often stateless²² meaning the output of an actor (i.e. the messages it sends) depends solely on its input (i.e. the messages it receives). While this has some additional benefits, it is not an absolute requirement for the actor model to work. In fact, an actor can have all kinds of local side effects and still be perfectly thread-safe, without having to resort to locking. The only place in an Erlang application where some level of synchronization is required is the message queues. However, these can be implemented lock- and wait-free, which allows for a high level of throughput.

When an actor sends a message to another actor, the runtime has to guarantee that the receiving actor can use the message without the risk of race conditions. This means that a message cannot contain references to mutable objects. It is therefore advisable for a language that implements the actor model to make all

²² There are several ways of writing stateful actors in Erlang, for example using closures or the Erlang in-memory database.

or at least most of its built-in types immutable. Where mutability is absolutely required, unique copies have to be made, before sending the types to other actors. While this can produce some overhead, it is absolutely required for guaranteeing thread-safety.

3.2.2. Programming with messages

Asynchronous message passing is a concept that can be difficult to grasp at first. Because communication between actors is asynchronous, an actor receiving a message cannot immediately respond to it. Instead, it has to send a message back to the sender who may not process it right away. Also, if an actor sends multiple messages to several different actors, it will receive their replies in a completely arbitrary order²³. This does not have to be a problem, but it requires an entirely different approach to concurrent programming.

To demonstrate how message passing differs from shared-state concurrency, we will now look at a familiar example (see Figure 17).

```
procedure Account(initial_balance)
  balance := initial_balance
  while true do
    message := receive
    switch message→type
      case "withdraw"
        balance := balance - message→value
      end case
      case "deposit"
        balance := balance + message→value
      end case
      case "balance"
        send {"balance", balance} to message→sender
      end case
    end switch
  end while
```

Figure 17: The bank account example implemented using the actor model

Here, we recreate the bank account example from chapter 2 using messages. To implement this example, we introduce a new keyword “receive” that fetches a message from the actor’s message queue. If the message queue is empty, the “receive” keyword blocks the actor and resumes execution, only after another message has arrived. The “Account” actor given in Figure 17 understands the three messages “withdraw”, “deposit” and “balance”. The “withdraw” and “depo-

²³ Erlang guarantees that messages sent from one actor to another will retain their ordering. However, when multiple actors send messages to the same other actor, the absolute ordering of those messages depends solely on which actor sends which message first.

“sit” messages do exactly as their names suggest, while the “balance” message can be interpreted as a request to the actor to send its current account balance back to the sender of the original message. An example of how to communicate with this actor is given in Figure 18.

```
account := spawn Account(100)
send {"withdraw", 50} to account
send {"deposit", 25} to account
send {"balance"} to account

while true do
  message := receive
  if message→type = "balance" then
    print message→value
  end if
end while
```

Figure 18: Sending messages to the account actor

Here, a new “Account” actor is created using the “spawn” keyword. We then continue by sending the actor a “withdraw” message, a “deposit” message and a “balance” message. Since the latter will cause the actor to send a message back to us, we have to enter a receive-loop ourselves and wait for that message to arrive. Since Erlang guarantees that messages sent to an actor in a particular order will also arrive and be processed in that order, the output of this program should be “75”, provided that no other actors exchange messages with the same account.

This example is much simpler than our previous bank account examples, because it involves only one account. This choice was made, because the actor model does not guarantee atomicity, even for related messages. In other words: when withdrawing money from one account and depositing it to another, for a short amount of time at least, the money will be in neither account. This is slightly less problematic in Erlang than it would be in a system that uses shared-state concurrency, because even without that atomicity, race conditions or lost updates cannot happen. Nonetheless, there are some use-cases where atomicity is absolutely required. Erlang addresses this issue with its integrated DBMS “Mnesia”. Mnesia is a distributed database that, like most other DBMS, supports transactions.

3.2.3. Fault tolerance

Erlang was invented in 1987, primarily for programming telecommunication switches. Back then, parallel hardware was not very common and it certainly was not used in the kind of switches that Erlang was designed for. Erlang’s use of the actor model therefore could not have been motivated by a need for parallelism. Instead, Erlang originally used “share-nothing” message passing to achieve fault tolerance.

When it comes to handling errors, the approach taken by Erlang is very different from that of other languages. The idea is not to prevent errors, but to allow individual actors to fail without compromising the stability of the system as a whole. If an actor fails, it can send a last message to a “supervising actor” and then terminate itself. The supervising actor may then restart the process, so that it can resume its work. Alternatively, an actor can monitor another actor and restart it, should it no longer respond to incoming messages. This form of error handling can be extended to multiple machines where one machine takes over the work of the other in case it should become unresponsive or crash.

Of course this approach works only because in Erlang, actors exist in total isolation and do not share any data. An actor that is terminated because of a program error does not produce dangling pointers anywhere else in the system. This has the positive side-effect that fault tolerance can be scaled by increasing the number of machines that stand by to take over the work of other machines, if they should fail. For this reason, Erlang is often used in applications where availability is critical and there are some Erlang applications that boast yearly uptimes in the region of nine nines²⁴.

3.2.4. Scalability

One of the challenges of the multi-core revolution is scalability (see chapter 4). Since the number of CPU cores is likely to increase significantly in the future, we need to find ways to design our applications so that they will benefit from these additional cores automatically and without us having to rewrite large portions of our code. In a traditional multi-threaded application, we tend to separate work into a fixed number of threads, but because these threads map more or less directly to hardware, increasing the number of CPU cores will only lead to an increase in performance for as long as the number of threads exceeds or is equal to the number of CPU cores.

Erlang addresses this issue by providing an abstraction (the actor) which is separated from the hardware. This allows the Erlang runtime to map actors onto CPU cores in whichever way it deems best. Of course this approach also does not scale indefinitely, because the runtime may still run out of work to distribute. However, actors tend to be much smaller than threads and they also tend to have just one purpose, which is why it not uncommon in an Erlang application to have hundreds or even thousands of actors. This is possible and even encouraged, because actors are much more lightweight than threads and their memory footprint is very small.

²⁴ The Ericsson AXD301 switch has shown to provide nine nines or 99,99999999% yearly uptime [17]. This translates to 31.5 milliseconds of annual downtime.

The actor model also scales well for when there is too much work for one machine to handle. Because actors communicate via message passing, different actors can also be running on different machines. Most of the time, this requires little or no changes to be made to the original application. Because of this property, Erlang has gained some popularity as a language for highly scalable server applications. It is used extensively by Ericsson, the company that Erlang originated from, but also by T-Mobile and several large websites and -services such as Facebook²⁵ and Amazon²⁶. Additionally, Google's MapReduce architecture, while not written in Erlang, uses pure message passing in a fashion that is very similar to Erlang [13].

3.2.5. Conclusion

Message passing concurrency has some significant advantages over shared-state concurrency. Not only does it guarantee thread-safety by default, it also provides powerful mechanisms for fault-tolerance and because actors are an abstraction that is independent from the underlying hardware or the operating system, they can easily be scheduled to run on any number of CPU cores or even on a cluster of networked computers. Also, because the actor model has a mathematical foundation, it is relatively easy to make assertions about the correctness of programs that follow this model.

Unfortunately, message passing is a concept that developers coming from the imperative school of thought tend to find difficult to understand. The fact that Erlang is a functional language and that its syntax resembles that of Prolog only adds to the problem. It is to be expected however, that message passing libraries will soon be available for many different programming languages. Unfortunately, because the type systems provided by most imperative programming languages do not enforce immutability, implementing a share-nothing message passing system on top of them can be a little cumbersome. This becomes apparent in libraries like Kilim²⁷, which, for understandable reasons, expose a lot of the complexity of guaranteeing thread-safety in such a language to the programmer [14]. If message passing concurrency is to enter the mainstream, we will therefore need better language support. Whether established languages like Java, C# or C++ will provide this support, remains to be seen.

²⁵ Facebook uses Erlang for their web chat (Source: Facebook).

²⁶ Amazon's online database SimpleDB is said to be written in Erlang [15].

²⁷ Kilim is a message passing framework for Java

4. Finding parallelism

An often underestimated challenge of the multi-core era is the question of how to find the parallelism in our applications and how to exploit it. But even if we manage to parallelize our applications, we may still not be able to keep all our CPU cores busy.

4.1. Amdahl's law

A dual-core CPU running at a 2 GHz clock-speed theoretically can execute twice as many instructions per second as a single-core CPU running at the same clock-speed. This also means that a program running on a dual-core CPU could theoretically be twice as fast as on a single-core CPU. In reality, this kind of linear speed-up is difficult if not impossible to achieve. While there may be several reasons for this, most importantly it is because there is a sequential part to pretty much every application.

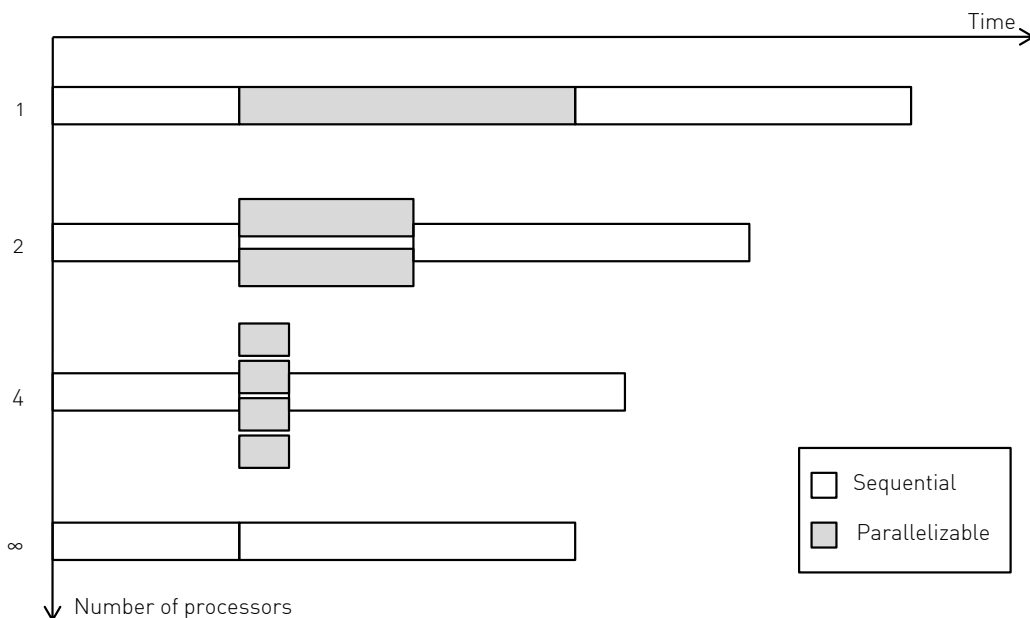


Figure 19: Performance gain from parallelizing parts of an application

Amdahl's law says that the performance of a parallel application is limited by its sequential parts (see Figure 19). In other words: if 50% of an application can run in

parallel, while the remaining 50% are sequential, the maximum performance gain that can be expected from running the application in parallel is also 50%. Because most of today's client applications are primarily sequential, a much greater performance gain can be achieved by parallelizing the sequential parts than by running the applications on a CPU with more cores.

This is the great challenge that the multi-core revolution confronts us with. No matter which abstractions for concurrency we choose, what matters in the end is how well we can exploit the parallelism in our applications.

4.2. Data parallelism

In many applications, a single operation has to be applied to a large set of data. By partitioning the data into several subsets, such operations can often be parallelized. This is especially true for operations that are referentially transparent. A good example of such an operation is summing up a set of numbers. By partitioning the set and applying the sum-operation to each subset individually, an almost linear speed-up can be achieved. Of course the individual sums have to be aggregated in the end, but apart from this, no synchronization is required.

While summing up numbers may be a trivial example, there are in fact much more complex algorithms that can be parallelized in a similar fashion. One popular algorithm that benefits greatly from parallelization is ray tracing. Ray tracing is a technology for generating photo-realistic 3D images. To generate an image using the ray tracing algorithm, a virtual ray is shot through each pixel to find its intersections with the geometry in the scene. By partitioning the image into blocks (called "buckets" in ray tracing terminology) and by rendering all blocks in parallel, ray tracing has become so fast that at least with simpler scenes and multi-core hardware, it can now be done in real-time²⁸.

4.2.1. Automatic parallelization

Because data parallelism, at least in the absence of side-effects, is relatively simple, efforts have been made to automate parallelization of data parallel algorithms or to at least provide convenient abstractions. In purely functional languages where side-effects are strictly controlled and often reflected in the type system, the compiler could theoretically parallelize certain algorithms transparently, without changing the semantics of the program. The same however does not apply to imperative languages. In an imperative language, there is little guarantee that a function is referentially transparent. Should a compiler attempt to parallelize a computation that has side-effects without taking the necessary pre-

²⁸ Real-time refers to the rendering speed required to achieve interactive frame rates (typically between 16 and 30 complete images per second)

cautions, the parallelized version might not yield the same results as the sequential version. For this reason, some language designers suggest that imperative languages should provide constructs to annotating functions that are pure in the mathematical sense, i.e. functions that do not have side-effects. Such “pure” functions could only call other functions which are themselves marked as “pure”. This would give the compiler a better understanding of the code, enabling it to perform more elaborate optimizations including automatic parallelization.

```
values := [1, 2, 3, 4, 5, 6, ... , 100]
result := []

pure function AddOne(operand)
  return operand + 1
end function

for i = 1 to 100
  result[i] := call operation(values[i])
end for
```

Figure 20: A function with the “pure” annotation

In the above example (Figure 20), a loop construct is given that repeatedly calls a function “AddOne” which is marked as “pure”. Using some of the same code analysis techniques commonly used for optimization purposes as well as the “purity” guarantee given by the “AddOne” function, the compiler can now not only determine that it is safe to parallelize the loop, it can also figure out how to partition it. It might then transform the loop into something like the following (see Figure 21):

```
pure function ProcessSubset(start, finish)
  for i = start to finish
    result[i] := call AddOne(values[i])
  end for
end function

number_of_cores := call GetNumberOfCPUCores()
subset_size = 100 / number_of_cores
workers := []
for i = 1 to number_of_cores
  start := (i - 1) * subset_size + 1
  workers[i] := spawn ProcessSubset(start, start + subset_size)
end for

call WaitForMultiple(workers)
```

Figure 21: The parallelized loop

The modified code now spawns one child process per available CPU core and then equally distributes the workload among these processes. It then has to block the parent process until all the child processes have completed their task.

Where the programming language does not provide constructs for guaranteeing purity, automatic parallelization is usually not available. However, programmers can still use the same mechanisms in a slightly more explicit fashion. The OpenMP API for example supports “parallelization through annotation” [15], where the programmer explicitly marks the loops and other constructs that he or she deems safe for parallelization. Additionally, library implementations of these mechanisms are available for many programming languages. Most notably these are the Parallel FX library for .NET and the Fork/Join framework available in Java SE 7. Of course without compiler- or runtime-enforced purity guarantees, it becomes the responsibility of the programmer to ensure that code passages annotated for parallel execution are in fact without side-effects.

4.3. Task parallelism

Because data parallelism tends to be quite easy to exploit, it is for the most part a well understood problem. However, in most applications and specifically those running on the client, data parallelism only makes up for a relatively small percentage of the total exploitable parallelism. The remaining parallelism is often found on the algorithm level and depending on the complexity of the algorithms, exploiting it can either be relatively easy, or extremely difficult. The primary problem with this form of parallelism is that decomposing algorithms into individual tasks is in itself a complex endeavor. This is especially true for algorithms where there are strong interdependencies between the individual parts of the computation. Oftentimes, tasks will have to synchronize with other tasks because they depend on their intermediate results. Parallelizing algorithms can therefore require a fair amount of analysis and optimization, especially because frequent synchronization between tasks is costly and can therefore degrade performance.

At this point, neither compilers nor runtimes possess the level of semantic understanding that would be required to automate the parallelization of algorithms. However, programming languages and libraries can provide abstractions that make parallelizing algorithms a lot more convenient. Most importantly, they can provide a simple mechanism for spawning and synchronizing tasks. The actor model presented in chapter 3 is one of those mechanisms. Additionally, some libraries like the herein before mentioned Parallel FX library provide similar mechanisms that are however more tailored towards imperative languages. But while these libraries can simplify the process of parallelizing and algorithm, the real work still has to be done by the programmer.

4.4. Scalability

At the time of writing, desktop CPUs typically had between two and four cores. Upgrading to such a CPU from an older single-core CPU may give the user the impression of better performance, because the additional cores can be used to run background processes²⁹ that would otherwise throttle the foreground processes. Even so, most multi-core CPUs are still largely underutilized, because typical client applications are either single-threaded or use only a small number of helper threads. While most software developers have not yet felt the need to parallelize their applications, CPU manufacturers certainly have. In the absence of parallel applications, it will become more and more difficult for CPU makers to convince consumers to upgrade their already underutilized CPU to one with even more cores. It is therefore not surprising that Intel as a hardware manufacturer is researching technologies like software transactional memory and has released a series of papers and articles on how to write parallel software.

This of course does not mean that parallelism is solely a problem of the hardware manufacturers. A piece of software that is performing poorly today is not likely to run any faster on tomorrow's hardware. This problem will grow as the number of cores increases: a single-threaded application running on a dual-core CPU uses 50% of the available resources, the same application running on a sixteen-core CPU uses only 6.25%. Given these numbers, it will become increasingly more difficult to justify any kind of performance problems in client applications.

Because the number of CPU cores in desktop PCs is likely to increase steadily over next few years, it is important to take this growth into account when writing parallel applications. Unfortunately, the common practice of using a fixed number of threads to perform all computations does not scale. We will therefore have to depart from using threads explicitly and move towards a more task-based system as illustrated in 4.3. A scheduler that becomes part of the application can then decide how many threads to spawn and how to distribute the tasks among them. This means that the application has to perform its own load-balancing, rather than leave it to the operating system. Fortunately, this can be done transparently either by the runtime or by specific library functions.

4.5. Running out of parallelism

With eight- and sixteen-core CPUs on the horizon, it becomes increasingly unlikely that we are going to be able to exploit this level of parallelism with standard application software alone. While there are some types of application such as im-

²⁹ Herb Sutter famously said that the main reason why adding more cores appears to make a computer run faster is because the additional cores can run all the spyware and adware.

age processing software that benefits greatly from multiple CPU cores, most other client software does not. An example that illustrates this problem very well is a word processor. While there is some level of parallelism inherent in a word processor, it is not nearly enough to keep four CPU cores busy, let alone eight or sixteen. Of course this is not really an issue, because word processors do not typically suffer from performance problems. Similarly, email clients, web browsers and music players all can be-, and often already are parallelized to some degree. But even if we run all of these programs at the same time, we would still not accumulate enough load to justify the use of an eight- or sixteen-core CPU.

In the past, when clock-speeds still doubled every two years, not only would existing applications become faster, software developers would also come up with new ways of exploiting these growing performance resources. For example, just a little more than a decade ago, it was absolutely unthinkable that even standard-definition video could be edited in real-time on anything less than a video editing workstation. Nowadays, consumer-PCs possess so much processing power, that they can be used to edit even high-definition video. Similar to this, PCs have routinely exceeded video game consoles both in performance and visual quality just shortly after their release. As the multi-core trend progresses, we will see similar improvements in existing applications as well as all new applications, that were previously too processing-intensive to run on a desktop PC. Even now, new applications are emerging that make heavy use of multi-core CPUs. One example is Microsoft's research project "Photosynth", which can produce a 3D visualization from a series of still pictures. Because this form of "computer vision" is so processing intensive, it could previously only be done on heavily parallel supercomputers. Another type of application that benefits greatly from multi-core CPUs is virtualization software. Using virtualization software, a user can run many instances of the same or of different operating systems on a single machine and at the same time. At least on the server-side, virtualization is often called the "killer-application" of the multi-core era. But it might prove useful on the client as well, for example as a security mechanism that allows users to surf the web in a completely isolated environment.

5. Conclusions

The multi-core revolution may very well turn out to be one of the greatest challenges of recent software history. Not only does it force us to write concurrent programs, something we have gone to great length to avoid in the past, it also puts everything we think we know about software architecture, testing and tooling into question. At the same time however, it lures us with the promise of almost unlimited performance resources and the chance to exploit them in innovative new ways. The question whether we even want multi-core CPUs has conveniently been answered for us by the hardware manufacturers. Multi-core is not coming, it is already here and now we have to make the best of it. What remains to be seen is if we are going to be open minded enough to shed old habits, adopt new paradigms and maybe even new programming languages. But even if we choose to ignore the multi-core trend, as many programmers are likely to do, at least in the short-term, change seems inevitable. Intel's 80-core Polaris CPU is the living proof of that.

Because CPU sales largely depend on the availability of applications that exploit their specific features, hardware manufacturers like Intel are now actively pushing developers into writing parallel applications. But programmers are notoriously cautious when it comes to adopting new programming paradigms and even though the multi-core revolution is already well underway, we will probably not see a large number of "multi-core-enabled" applications any time soon. Language- and library designers are in part to blame for this situation, because not only have they missed out on the multi-core trend for many years, many of them have still not given any definitive answers as to how they will evolve their languages to better support parallel programming³⁰. But while the picture so far may seem rather bleak, the reality is not all bad. Many developers are now actively debating on the internet, what they think parallel programming should look like in the future. Also, functional programming languages like Erlang or Haskell are enjoying a surprising renaissance and a small but growing group of programmers has given up imperative programming altogether, in favor of a more declarative programming model. This trend is reflected upon the mainstream languages in

³⁰ The appendix of this thesis may give a better insight into what the future of parallel programming might look like.

interesting ways. For example, both C# and the next release of C++ feature higher-order functions and closures.

But while a few “enlightened” programmers are already exploring alternative concurrency models, the real challenge will be to convince the remaining programmers to start developing parallel applications. This can only succeed if we can find abstractions that are easy and familiar and as much as we like to deny it: syntax does matter. Erlang may be the perfect language for the multi-core era, but because it looks like Prolog, its chances of becoming the next big mainstream programming language are close to zero. Software transactional memory on the hand may possess all the qualities of a concurrency model that the mainstream can accept. But because it is built on top of the shared-state concurrency model which over the years has acquired quite the bad reputation, some developers feel that it too is inadequate for the kind of parallelism we will need in the future. This feeling is reinforced by the fact that the multi-core revolution may be a once in a lifetime chance to do away with outdated programming paradigms.

The multi-core revolution is correctly labeled a “revolution” because it puts everything we know about programming into question. Of course this raises all kinds of fears and keeps many developers from seeing the great potential in this trend. In that respect, the multi-core trend may be a little bit like the Web 2.0, where it took a handful of pioneers to pave the way for those who followed. But seeing how there already is a great amount of interest in this technology, it is now only a matter of time until it will take off.

References

- [1] Sutter, Herb. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal. 2005, 3.
- [2] Intel CPUs, an overview. Titan city. October 2006. <http://titancity.com/articles/intel.html>.
- [3] Myers, David G. Psychology, Eighth Edition, in Modules. s.l. : Worth Publishers, 2007. 978-0716779278.
- [4] Wilson, Timothy D. Strangers to Ourselves: Discovering the Adaptive Unconscious. s.l. : Belknap Press, 2002. 978-0674013827.
- [5] Leveson, Nancy G. Medical Devices: The Therac-25 Accidents. 1993.
- [6] Hoare, C. A. R. Communicating Sequential Processes. Prentice Hall International, 1985.
- [7] Gilheany, Steve. Evolution of Intel Microprocessors: 1971 to 2007. Berghell Associates. 28. March 2002. <http://www.berghell.com/whitepapers/Evolution%20of%20Intel%20Microprocessors%201971%20to%202007.pdf>.
- [8] Harris, Tim, et al. Composable Memory Transactions. Cambridge : Microsoft Research, 2006.
- [9] Peyton-Jones, Simon und Harris, Tim. Programming in the Age of Concurrency: Software Transactional Memory. Channel 9. 1. September 2006. <http://channel9.msdn.com/shows/Going+Deep/Programming-in-the-Age-of-Concurrency-Software-Transactional-Memory/>.
- [10] Langley, Adam. Software Transactional Memory. Google Video. 28. February 2007. <http://video.google.com/videoplay?docid=5442492239822665187>.
- [11] Armstrong, Joe. Armstrong on Software: Erlang & SMP. Google Video. 26. June 2008. <http://video.google.com/videoplay?docid=-1363340548829250196>.
- [12] Armstrong, Joe. Programming Erlang, Software for a concurrent world. The Pragmatic Programmers, LLC, 2007. 1-9343560-0-X.
- [13] Dean, Jeffrey und Ghemawat, Sanjay. MapReduce: Simplified Data Processing on Large Clusters. 2004.

- [14] Srinivasan, Sriram und Mycroft, Alan. Isolation-Typed Actors for Java. Kilim. [Online] 2008. <http://www.malhar.net/sriram/kilim/>.
- [15] OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. 2008. <http://openmp.org/wp/>.
- [16] Transactional Memory in Sun's "Rock" processor. System News Volume 114, Issue 4. 20. August 2007. <http://sun.systemnews.com/articles/114/4/news/18516>.
- [17] Intel Corporation. Intel Architecture Software Developer's Manual Volume 2. 1999.
- [18] Ying, Charles. What You Need To Know About Amazon SimpleDB. Inside looking out. 13. December 2007. <http://www.satine.org/archives/2007/12/13/amazon-simplydb/>.
- [19] Armstrong, Joe. Making reliable distributed systems in the presence of software errors. Stockholm, Sweden, 2003.

Appendix

While doing the research for this thesis, I realized very early on that programming languages will ultimately determine how well we will be able to express the parallelism in our applications. When I came to that conclusion, I was very keen on learning what some of the most well-known language designers thought about the issue and how they planned on dealing with it in their programming languages. I therefore sent emails to Anders Hejlsberg (C#), Joe Armstrong (Erlang), Dr. Alan Kay (Smalltalk), Guido van Rossum (Python), Dennis Ritchie (C), Simon Peyton-Jones (Haskell), Dr. Bjarne Stroustrup (C++), James Gosling (Java), Brendan Eich (JavaScript) and Yukihiro Matsumoto (Ruby), in which I asked all of them the following two questions:

“With the emergence of the first multi-core computers and many-core computers on the horizon, programmers will have to find better ways of expressing the parallelism in their applications. The traditional model of threads and locks is known to be error-prone and is inadequate for the kind of parallelism we will undoubtedly need, to fully benefit from the vast CPU resources available to us now and in the future. How do you think programming languages should deal with this issue? Can we find better ways of expressing parallelism in the presence of side-effects or are we inevitably on a road towards a more declarative programming model?”

I received a surprising number of replies, all of which are quoted below:

Guido van Rossum, author of Python

“I'm not sure how to answer it [the question], except that it is my strong conviction that threads and locks are not the answer. They are based on a “flat” memory model where accessing any byte in memory has the same cost. That already is vastly wrong with the several levels of memory caches; with multiple cores it will become even more wrong, as maintaining cache consistency becomes more expensive with the number of CPUs. I expect that a more “process” based model (whether based on real processes or an abstraction like found in Erlang) will eventually win out, with message passing instead of locks and shared memory as the primary communication primitive. Beyond that, it's anybody's guess; I expect it will be many years before this becomes relevant to Python.”

Dr. Bjarne Stroustrup, author of C++

“I wouldn't want to write an application directly using thread and locks. That's about the worst way known. However, most other models are specialized in that

they don't cover all needs - languages with specific language features supporting a higher-level have found that. I think that the solution to this dilemma is to provide a set of concurrency primitives (threads, locks, atomic types, etc.) and build the higher level models as libraries on top of these primitives. That's the approach taken by C++0x. For C++ the threads are especially important because those are needed for complete access to system resources. In general, if we can express something declaratively, we should. However, in the field of concurrency we are far from that point. For higher-level models, I'm in favor of generating many small "tasks" (far smaller than threads) that can be executed in parallel as processors and memory allows. I also like message queues to eliminate shared-memory problems."

Joe Armstrong, author of Erlang

"No [I don't think we can find better ways of expressing parallelism in the presence of side-effects] - certain things like transaction memories help the situation, but still represent single points of failure and change the failure characteristics of a program. 30 years of research in adding parallel constructs to imperative languages has not come up with much so I'm pessimistic. Yes [we are indeed on a road towards a more declarative programming model] - FPLs¹ (pure) have no mutable state and thus state does not have to be protected with locks. However, it's still a research problem to decide how to partition a (say) data flow diagram onto a multi-core. Erlang makes life easier since the programmer has already decided on what the granularity of concurrency is (i.e. the process). Erlang processes that do not use ETS² are automatically thread safe which is very helpful."

Simon Peyton-Jones, co-author of Haskell

"Concurrency is complicated and we need more than one parallel-programming paradigm. I see three good ones for parallel FP³: Semi-implicit parallelism with `par` and `seq`⁴, explicitly forked threads synchronized with transactional memory and nested data parallelism⁵."

¹ Functional programming languages

² Erlang term storage - Erlang's in-memory database

³ Functional programming

⁴ In Haskell, `par` and `seq` are means of annotating code for parallel and sequential composition

⁵ Nested data parallelism is the application of parallel operations to bulk data as opposed to the application of sequential operations to bulk data.

Larry Wall, author of Perl

"That's a deep question, and no one has the perfect answer to it, not even the functional folks. Everyone is starting to worry about it, though. (I just attended a concurrency summit a couple of weeks ago with some of the Big Names, and we talked about it all day without coming to any kind of resolution.) Well, actually, not everyone is starting to worry about it. I've been worrying about it for many years now...

In one sense, I think we're inevitably on a road towards something resembling functional programming. However, I think the current approaches to concurrency taken by the functional programming community are far too rigid, brittle, and esoteric to scale well to the masses.

What I think we're going to see is more thinking about the granularity of this notion of side effects. As a language designer, I'm interested in looking at constructs that promise computational independence without necessarily promising strict absence of side effects. So in the design of Perl 6, we're introducing various constructs that make some kind of promise about computational independence, among which we have:

$\downarrow ==$ and $== \uparrow$ "feed" operators

These are essentially object pipes. As with Unix pipes, the code invoked on one end of the pipe runs independently of the code on the other end.

hyperoperators like $\@a \gg \ll \@b$

Hyperoperators are explicitly vectorizable with run-to-completion semantics.

junctional logic like $\$a == 1 | 2 | 3$

Junctions support data parallelism where not all threads have to run to completion, but you can bail out as soon as you can prove or disprove something. Junctions are logically evaluated in parallel, so the computer is free to optimize the tests into whatever order makes sense to it.

contend {...}, maybe {...}, and defer

These support software transactional memory; we're trying very hard to avoid explicit locking in the new design. This tends to be a rather philosophical argument these days, but in my mind, the eventual advantage of STM will be that you can virtualize time. Hard locks are not virtual with respect to time, since they're anchored to "now". Virtual time allows two processes to negotiate a way forward where both views can remain consistent without specifying how the constraints on consistency are mapped to real time. Open issue: how to extend the concept outside the domain of mere memory access.

threads vs. events

We're also thinking about how to install a scheduler within the process such that the programmer can simultaneously have a high-level transactionally threaded view, while underneath everything is run by asynchronous events in a more Erlangish fashion. Neither view is complete in itself, and each view is ideal for a different set of problems.

unthrown exceptions

When you're doing things in parallel, exception handling must become a data-flow concept, not a control-flow concept. Perl 6 has the concept of unthrown exceptions that can stand in for any OO-based return value. So a vector operation might produce a list of results, some of which indicate failure, without blowing up the whole parallel computation. It can be really embarrassing if your rocket keeps throwing away 999 good calculations because 1 out of your 1000 sensors is bad. Think of this as an extension of the IEEE NaN concept.

That's an example of a more general principle: it's just really important to "limit the scope of the damage" continually. Historically this principle has mostly been applied to declaring your lexicals in the smallest practical scope, but the principle needs to be applied much more pervasively. We know that globals are evil, but they keep sneaking into our designs in various subtle ways. And any info that is attached to an object that is "too global" will have similar effects. Even functional programming language typically fall into the trap of hiding lots of computational state in the stack, which is much more global than any individual function call.

And, of course, objects are completely stateful, and every time you screw up your class/parent/delegate hierarchy, you're making similar mistakes of having something a little too global. So another thing we're trying to be very careful about in Perl 6 is to distinguish mutable classes from immutable "roles" (based on Smalltalk Traits).

Basically, we're borrowing some good ideas from FP without going all the way. You don't have to go all the way to the FP mindset and make all data immutable, but you do at least have to keep track what is mutable and what is not. More generally, we'll be making better use of our type systems to do late-binding run-time dispatch, relying on the type system to set up constraints without nailing everything down at compile time. And in any case, I think we will definitely be seeing more of a shift towards declarative programming, where you just tell the computer what you want, not how to do it.

The real trick is how to do all this in a language that remains accessible to inexperienced programmers. The problem with most computer language designers is that they're too smart to remember what it was like to be stupid."

Yukihiro Matsumoto, author of Ruby

“I see two possibilities: concurrency savvy language like Erlang will become dominant or, concurrency supporting systems (or libraries) on top of more traditional languages e.g. MapReduce will. I am not sure which future is more probable. But I expect the latter, and we already started working on it.”

Roberto Ierusalimschy, author of Lua

“You may already know that I am absolutely against multithreading (shared memory + preemption), except for very low-level programming (e.g., when writing the kernel of an OS):

We did not (and still do not) believe in the standard multithreading model, which is preemptive concurrency with shared memory: we still think that no one can write correct programs in a language where $a = a + 1$ is not deterministic.⁶

It is not only inadequate “for the kind of parallelism we will undoubtedly need”; it is inadequate for the kind of parallelism we already have. The only reason people still use it is because current software practices are somewhat complacent with bugs.

In my view, we do not have to give up side effects, just to give up shared side effects. For instance, for each relevant data structure in your program, put one process to keep it; all operations over this data structure become messages to this process to perform the operations.

One problem we have today is that all software practices and all hardware development goes toward “standard” multithreading. So, we do not have the techniques and shared knowledge to develop other forms of parallelism. Moreover, hardware is not optimized for it. (For instance, without shared side effects, cache synchronization would be much cheaper; but with current hardware we pay that price even without using it.) To defend other forms of concurrency with numbers (benchmarks), we must compensate for this bias in current software and hardware.”

⁶ This is a direct quote from the research paper “The evolution of Lua” by Roberto Ierusalimschy et al.