Evaluation of Multi-Core Programming Models

Charl de Leur
c.deleur@student.utwente.nl

ABSTRACT

With the advent of multi-core processing in personal computing systems, the demand for concurrent software has increased far beyond its original niche. The conventional model of lock-based concurrency for writing such software is difficult in use, error-prone and does not always lead to efficient use of the available resources. In this paper we shall evaluate several alternative models, namely the *Actor Model*, the *Join-Calculus*, *Software Transactional Memory* and *Runtime Scheduling* and compare them to the conventional one, to determine whether they provide superior means of writing concurrent software.

Keywords

concurrency, threading, programming

1. INTRODUCTION

Due to the rise of multi-core processors in modern personal computing systems, concurrent programming is no longer a niche that only programmers of software used on servers or in HPC[1]-situations encounter. The methods of writing concurrent programs have however remained largely the same.

This is in part due to the hardware used and in part due the operating systems used, that provide programmers with the means to use said hardware. All modern systems use the concept of interleaving: independently and possibly concurrently running computations [4]. This allows programmers to make use of the systems resources and to allow for a reactive system under modern workloads. However, interleaving by itself is not sufficient to write functional concurrent software. What is missing is a method to communicate and orchestrate several computations when they are to share information or when they are contending for resources. The machine and the operating system only provide the most basic of atomic instructions to allow for these operations, therefore models have been implemented on top of these instructions to allow for a practical use of the available resources.

Nowadays most programmers are taught and make use of the same model for concurrency, namely *lock based concurrency*, a model that uses a mutable shared state for communication between processes, and many are lead to belief that it is the only one. Difficulties [45] with this now conventional model have led to a resurgence in research towards alternative models that might have less of these problems and that may be more efficient in their use of the available resources. First we shall look at some of these difficulties to see how alternative models compare to the conventional one in Section 2 and at the related work on the subject in Section 3, followed by the problem statement and research questions in Section 4 and the research approach in Section 5. The actual evaluation can be found in Section 6 and the conclusions in Section 7.

[1]High Performance Computing using supercomputers and clusters

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission.
11th Twente Student Conference on IT, Enschede June 29th, 2009
Copyright 2009, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science

2. BACKGROUND

2.1 Problems of Mutable Shared State Concurrency

We can identify six generally occurring problems with mutable shared-state concurrency, which is the most common variant of concurrency in use.

- *Mutual exclusion*
  For processes in a concurrent program to work correctly with shared resources, mutual exclusion is required. If this would not be the case, it would be possible for one process to read a resource that is still being modified by another process and for it to cause other unwanted scenario's. This results in broken invariants which can lead to unexpected or unwanted states [4].
- *Race conditions*
  Race conditions occur when the output of a process is dependent on the timing of another process or event. This causes problems when computations aren't executed in proper sequence or timing, resulting in an unexpected state. This leads to extremely difficult debugging, because improper timing might only occur in a small subset of possible computations [38].
- *Deadlock*
  In a deadlock the program hangs in an invalid state due to processes waiting on each other, barring progress. Deadlock occurs when the so-called Coffman conditions occur simultaneously [13].
  1. Tasks claim exclusive control of the resources they require ("mutual exclusion" condition).
  2. Tasks hold resources already allocated to them while waiting for additional resources ("wait for" condition).
  3. Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion ("no preemption" condition).
  4. A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain ("circular wait" condition).
- *Starvation*
  Starvation occurs when a process is denied access to a resource due to another process continuously taking exclusive control of said resource. A special case of starvation is a *livelock*, in which all processes continue to change state, but no overall progress is achieved. Starvation is generally illustrated with the *dining philosophers* problem [12].

- *Nondeterminism*
  Because of the interleaving of computations of multiple processes in a concurrent program, the resulting program ends up nondeterministic even if the processes being interleaved are deterministic to begin with. This makes it hard to reason about the program and it is one of the common causes of race conditions [31].
- *Lack of predictable composability*
  Individually correct pieces of concurrent software cannot be guaranteed to produce a correct program when composed. Additional work is required to make the composition valid, especially when the composed processes wish to access the same shared resource.

2.2 The conventional approach

The conventional approach to threading uses two constructs: semaphores and monitors, both built on the locks construct provided by operating systems.

Semaphores [15, 44] are a traditional synchronization primitive used in lock based concurrency, these days mostly used as building blocks for higher level solutions or in programming languages that do not support more complex forms of locking. Most operating systems provide the semaphore as primitive synchronization mechanism. They can be used to solve the problems of mutual exclusion and race conditions (famously so in *dining philosophers* [4]), but are still prone to the other problems mentioned in 2.1.

Monitors [25] are a more advanced synchronization primitive, allowing mutual exclusion to methods within the monitor and waiting and signaling on conditions. This can be used to determine whether another process is allowed to access a resource, or whether it must wait until it is signaled, because the resource is in use by another process. Monitors allow easier reasoning about concurrency than semaphores, but still have the same problems.

While proven solutions to problems of threading using these tools exist, they are hard to implement correctly. Changes to the software engineering process using these tools alone will not be enough to make their use much more practical [31].

2.3 Alternate approaches

Some of the leading alternate models for concurrent programming are:

2.3.1 The Actor Model

The *Actor Model* is a model stemming from AI-research in which the entire system is represented by so called Actors [3]. It uses message passing to allow communication between the various `Actors` that make up a system. `Actors` are generally allowed to do three things on receiving a message :

- Send a finite number of messages to other `Actors`.
- Create a finite number of new `Actors`.
- Specify the behavior to be used on the next message.

These actions are all executed concurrently, making the *Actor Model* inherently concurrent.

Compared to the conventional model, the *Actor Model* has the following qualities:

- *Mutual exclusion*
  Because of the message passing nature of the *Actor Model*, there is no mutable shared state, so mutual exclusion is much easier to handle, by wrapping the shared resource actor in a so called `serializer`, which queues messages and limits access to the shared resource. All access to the resource must happen via this `Actor`, so problems are less likely to occur [3].
- *Race conditions*
  Race conditions are still a likely problem.
- *Deadlock*
  Because the *Actor Model* is essentially lockless, deadlock should not occur, although there are scenario's where similar problems occur [9, 26].
- *Starvation*
  Starvation is still a likely problem.
- *Nondeterminism*
  The nature of message passing in this way and using rendezvous or Petri Nets [40] minimizes the amount of nondeterminism [31].
- *Lack of predictable composability*
  Because there is no shared state `Actors` generally have less composability problems, but protocol issues for instance make it so that composition is not guaranteed to be successful [1].

Added problems are messaging overhead, the lack of inheritance in `Actors` and the requirement of unbounded message boxes on the actors to allow for weak fairness.

The premier implementation of the *Actor Model* is *Erlang* [30, 2], an alternative one is *Scala Actors* [22, 23].

2.3.2 The Join-Calculus

The *Join-Calculus* is a model based on the π-Calculus [37], a *Process Calculus* designed to formally model distributed systems. The *Join-Calculus* approach to concurrency strongly resembles that of the *Actor Model*, with the exception that the *Join-Calculus* features anonymous processes instead of named `Actors` and instead of using the name of an `Actor` as an address to send messages to, shared named channels or ambients [8] are used. The *Join-Calculus* has the same characteristics as seen before with the *Actor Model*. It is also the case that the *Actor Model* can be modeled in the *Join-Calculus* and vice versa [17, 18].

The premier implementations are the join calculus language [41] and JoCaml [34], both of the ML-subset of functional languages, due to the easy mapping between the *Join-Calculus* and such languages. Alternatives are Cω [5], boost.join [32] for C++ and Join Java [28].

2.3.3 Software Transactional Memory

Software Transactional Memory is a model based on transactions, as also used in database systems. Concurrency is handled by letting threads run their transactions simultaneously, verifying the result and comitting if shown that no conflicts have occured. If unsuccessful the transaction is either aborted or retried [43]. Compared to the conventional model, the *Software Transactional Memory* has the following qualities:

- *Mutual exclusion*
  Mutual exclusion is unnecessary due to the commit/abort system of transactions.
- *Race conditions*
  Transactions eliminate race conditions.

- *Deadlock and Starvation*
  Transactions tend to eliminate unintended deadlocks and starvation [31, 16].
- *Nondeterminism*
  Transactions are highly nondeterministic, which makes them useful for intrinsically nondeterministic situations, but less so for determinate concurrent interactions. [31]
- *Lack of predictable composability*
  Transactions are not composable by nature [31], but can be made so [24].

This model is a favorite among many scholars, although problems have been noted [39, 10]. The premier implementation is in GHC Haskell [24], but due to the relative popularity [16] of *Software Transactional Memory*, many implementations exist.

2.3.4 Runtime Scheduling

A model in which a runtime scheduler divides workloads based on annotations in the sourcecode, added syntax or *pragma directives*. Compared to the conventional model, the *Runtime Scheduling* has the following qualities:

- *Mutual exclusion*
  Mutual exclusion is handled by defining which parts of the program are shared resources.
- *Race conditions*
  Race conditions are still a problem.
- *Deadlock and Starvation*
  The runtime scheduler allows for a fair and optimal use of resources and use of runtime information to prevent deadlock and starvation.
- *Nondeterminism*
  The use of runtime scheduling increases the amount of perceived nondeterminism, unless the heuristics of the scheduler are clearly understood.
- *Lack of predictable composability*
  Because it is known which parts are to be used concurrently, but the actual scheduling happens at runtime, programs using runtime scheduling can be safely composed.

The use of a runtime scheduler does carry some overhead.

Due to a lacking formal definition of runtime scheduling, wildly differing implementations of this model exist. Some of the more popular ones are *Cilk* [6], *Intel®threading building blocks* [49] and *OpenMP* [20].

Cilk is a language for multithreaded parallel programming, developed at MIT. It uses new language constructs for the definition of concurrent sections of code and uses a process called resource stealing for fair runtime scheduling. *Intel® threading building blocks* uses a C++ template library to create constructs such as `parallel_foreach` to define tasks that can be scheduled at runtime. *OpenMP* uses *pragma directives* to annotate the code.

3. RELATED WORK

The problems with the conventional lock-based model for concurrency are well known and often mentioned [31, 4, 38, 13, 12, 29]. Competing models are sometimes compared to the conventional one [31], but benchmarks are seldom done and then only within the confines of a single model [21, 39] or programming paradigm [7].

4. PROBLEM STATEMENT

The current conventional lock based approach to multithreading is difficult in use, error-prone and does not always lead to efficient concurrent programs. Several alternative models have been proposed, some of which mentioned above, which theoretically should have less of the general problems of mutable shared state concurrency. The question is whether these models can be practically used and if they actually compare favorably to the conventional approach.

The goal is thus to perform an empirical analysis of the aforementioned models and to compare them with the conventional model.

4.1 Research Questions

Following this, the main question to be answered would be:
do different approaches to concurrency compare favourably to the lock based one?
To quantify the answers to this question, we use the following subquestions:

1. How difficult are they to use?
2. How fast are they in comparison?

5. RESEARCH APPROACH

To answer the questions raised in the previous section, a testsuite consisting of known standard problems in concurrency, such as *readers/writers*, *producer/consumer*, *dining philosophers* [4] or the *Santa Claus Problem* [48] has been implemented using the lock based model and each of the alternate models.

Because these problems were all thought of with lock-based solutions in mind, the suite could be considered unfair at this point. To alleviate that concern, each model will also be separately evaluated on problems defined to showcase its strengths and weaknesses.

Furthermore we shall have a look at how performant each implementation of a model is and finally attempt to rank them.

6. EVALUATION

In our evaluation of the different models we shall first look at the difficulty of use of the model, using the general problems of mutable shared state concurrency mentioned before as a guideline. Most of the evaluation will be done using the relatively simple bounded buffer problem, expanding to different examples where necessary. Secondly we shall compare the relative performance by taking the standard lock based sample of the bounded buffer as our baseline and then seeing if the other models are faster or slower. Because we are comparing vastly differing implementations, native, virtual machine based and interpreted, only a general view of performance will be used instead of a broad and strict set of benchmarks. Lastly we shall give some notes on efficiency.

6.1 The Lock-Based Model in Java

Java will be the baseline language in our comparison as it offers full and mature support for the lock-based model of concurrency and does so in a clean and relatively straightforward way.

6.1.1 Usability

- *Mutual exclusion*
  The *Java* sample program uses *Monitors* and *Conditions* to guarantee mutual exclusion. We use the monitors explicitly by using the primitives in `java.util.concurrent` instead of the more common `synchronized` blocks, to better

approximate the classic monitor. The locks and conditions are instantiated as objects in the standard OOP style, as shown in listing 1:

Listing 1: Locks & conditions in Java

```
private final ReentrantLock lock = new
    ReentrantLock();
private final Condition notFull  = lock.
    newCondition();
private final Condition notEmpty = lock.
    newCondition();
```

For mutual exclusion only the `ReentrantLock`, which is the actual monitor object is necessary. By taking a lock on the monitor object, you make sure that only the current process will be able to enter the critical section, assuming that all processes have to go through the lock. Such a protected critical section, as also used in the bounded buffer program, is demonstrated in listing 2:

Listing 2: A critical section in Java

```
lock.lock();
try
{
  [..]  //critical section
}
finally {
  lock.unlock();
}
```

We make sure that all processes actually try to acquire the lock and not sneak around it by putting in the `get` and `put` methods of a separate buffer object. This lets the buffer decide when to block producers or consumers instead of those processes themselves.

- *Race conditions*
  The monitor we use to guarantee us mutual exclusion also prevents two threads from racing, as only one is allowed into the critical section. If we would remove the protection from the get-method of our buffer, assuming that it would still block on an empty buffer, and start two consumers, the behaviour of our program would become highly nondeterministic due to the racing consumers.

- *Deadlock*
  The condition variables allow us to block threads by waiting on them inside the monitor block. Modifying our program to deadlock, while using monitors, is as simple as forgetting a signal. The signal/wait as used in Java in our sample bounded buffer can be seen in listing 3:

Listing 3: Signaling in Java

```
//put
while (count == buffer.length) notFull.await();
[..] // buffer
notEmpty.signal();

//get
while (count == 0) notEmpty.await();
[..] //unbuffer
notFull.signal();
```

- *Starvation*
  Our sample bounded buffer is free from starvation due to the bounds on the buffer and the use of only one producer and consumer thread. Once we up the amounts of consumers however, there is the possibility of starvation as a random thread is allowed execution when a condition is signaled in Java. A solution for this is the use of a fair `ReentrantLock` (by instantiating it as such), which will guarantee fairness by always favouring the longest waiting thread. This will unfortunately generally result in slower overall throughput.

- *Nondeterminism*
  The timeslicing of the JVM[2] and the randomness when awaking threads make this implementation of a bounded buffer highly nondeterministic. This is not much of a problem in our sample program, as the bounds on the buffer and the small serial critical sections make it relatively easy to reason about. However, it is easy to see how this escalates in larger programs.

- *Lack of predictable composability*
  In the bounded buffer sample composability is not much of an issue as both producers and consumers can be added without issue.

- *General remarks*
  The processes in *Java* are blocked by calls on the buffer, instead of blocking themselves as in the erlang sample. The bounds on the buffer are enforced by blocking threads on condition variables, not by expensive busy wait loops or similar constructs.

6.1.2 Performance & Efficiency

In *Java* the Sun JVM used with our tests uses kernel threads, generally working best when not too many threads are spawned. While kernel threads are often said to be slow in creation, the JVM allows thread creation/destruction in the order of 10^4 per second, more than sufficient for most applications.

It is also generally assumed that synchronized methods in java are especially slow, but in practice this is mostly false [19].

The critical section model of programming is a pessimistic form of concurrency control, meaning that it will always block, regardless of actual contention. This has the tendency to cause slow performance because only one thread is allowed in a critical section. With overly large critical sections or sloppy synchronization, performance suffers.

6.2 The Actor Model in Erlang

For the *Actor Model* we shall use *Erlang*, a concurrent functional language designed by Erikson.

6.2.1 Usability

- *Mutual exclusion*
  Mutual exclusion in the *Erlang* sample is handled by having a buffer process which is responsible for keeping the state of the buffer by calling itself recursively. The buffer process handles two asynchronous messages, `get` and `put`, which are serialized and buffered by the *Erlang* runtime, so that only one is handled at the time. Listing 4 displays the buffer process in *Erlang*:

Listing 4: A buffer process in Erlang

```
buffer(Contents, Capacity) ->
  receive
    {put, Producer, Item}
     when Capacity > 0 ->
      Producer ! {produce, self()},
      buffer(Contents++[Item], Capacity-1);
    {get, Consumer}
     when Contents =/= [] ->
```

[2]Java Virtual Machine

```
9       Consumer ! {consume, self(), hd(Contents)},
        buffer(tl(Contents), Capacity+1)
11    end.
```

The `buffer` process is a recursive function with 2 parameters, `Contents`, which is the current contents of the buffer and `Capacity`, which is the remaining capacity of the buffer. The `receive` handles two messages: `{put, Producer, Item}` and `{get, Consumer}`. The first message specifies an addition to the buffer, with the address of the producer and the item to be added, guarded by the fact that there should be room for the item. The second message specifies removal, with the address of the consumer and a *guard* stating that the buffer should not be empty. On a `get` or a `put`, a message is sent to the caller, returning the asked item or asking for new one and `buffer` calls itself recursively with the updated state.

- *Race conditions*
  We have mentioned before that that race conditions are still a likely problem in the *Actor Model*, but exactly why might not immediately be apparent in a system that is supposedly without shared state. The problem is that while *Erlang* doesn't share state in the traditional sense of mutable variables, it is quite possible to emulate mutable state, for instance as in our buffer to keep track of the contents. This manner of keeping state once again allows race conditions, for instance in the manner of James Iry's Sock Machine [27].

- *Deadlock*
  Our example bounded buffer is free of deadlock as one would assume with a "lockless" model such as the *Actor Model*. However, because *Erlang* blocks threads waiting for messages, by blocking when no messages are queued or the guard on the patterns do not match, it is actually quite easy to force a deadlock as listing 5 demonstrates:

Listing 5: Deadlock in Erlang

```
1  receive
     {From, foo} ->
3      From ! {self(), bar};
   end
5  [..]
   receive
7    {From, bar} ->
       From ! {self(), foo};
9  end
```

Two processes in a deadlock, each simply waiting on a message from the other. Waiting on a message has the same negative effects as waiting on a condition variable in *Java*, with or without explicit locking. In our buffer example then, as an analog to forgetting to signal in *Java*, it is possible to deadlock our producer by neglecting to send a message for production of another unit.

- *Starvation*
  Starvation is once again not an issue in our example of one producer and one consumer, but in the case of many consumers to one producer it becomes an issue of the fairness of the scheduler in *Erlang*. Another issue is the use of pattern matching on messages, as the messages higher up in the matching chain are given precedence (in our example `put` is thus given precedence over `get`), this is not an issue as we have an upper bound on production which guarantees a turn for the consumer, but in an unbounded system it would probably be preferable to take the last message and then match instead of matching over the buffered messages implicitly giving precedence.

- *Nondeterminism*
  The asynchronous messaging used in our sample program is highly nondeterministic by nature, this could be mitigated by using synchronous or rendezvous communications, but this takes away some of the elegance the solution. Some of the nondeterminism inherent with asynchronous communications is lessened by the implicit use of serializers in *Erlang*, making sure the messages arrive in proper order.

- *Lack of predictable composability*
  Generally *Erlang* systems appear to be highly composable, however, protocols used, especially in larger systems, have to be designed with composability in mind. For instance, a protocol requiring messages to carry a strictly incrementing sequence number without gaps can be implemented with a simple counter when communicating with one thread, but will fail when another is added.

- *General remarks*
  It is clear that due to the message passing nature of *Erlang*, the importance of a proper protocol for your messages cannot be understated. The issues mentioned before mostly tend to be protocol issues. These are easily avoided in simple cases such as our bounded buffer, but with more complex protocols they tend to be much harder.

  Compared to the *Java* implementation it is interesting to see that in this case our producer and consumer processes tend to block themselves instead of being blocked by an external piece of code. The *Erlang* code is also very short, but remains quite readable and the language is quite easily learned.

6.2.2 Performance & Efficiency

Erlang uses a concurrent runtime supporting SMP[3], allocating a single kernel thread per processing unit by default. The runtime then schedules the lightweight user threads created with the `spawn`-command by mapping them to the kernel threads. User thread creation in *Erlang* is extremely fast (in the order of 10^5 per second) on a modern PC.

Essentially the *Erlang* SMP scheduler works by having a global queue protected by locks which is used by the schedulers spawned by *Erlang* to get their units of work, consisting of mostly processes. This presents somewhat of a bottleneck. Also the memory allocator locks, which hampers performance as well. Both of these problems are scheduled to be solved in a future version of Erlang.

As it stands the *Erlang* runtime scales quite well on 2-4 processing units, and less on more, with basic performance of the interpreted *Erlang*-code being very slow and *HiPE*-compiled [42] code being substantially faster, but still not optimal.

6.3 The Join-Calculus in JoCaml

Our *Join-Calculus* sample bounded buffer will be implemented in *JoCaml*, a functional language derived from *OCaml* with added *Join-Calculus*-primitives. Like *Erlang* it uses message passing concurrency, only instead of defining endpoints and addressing those, you define channels for communication, which can either be synchronous or asynchronous. You then use so called *join patterns* on those channels to handle messages.

6.3.1 Usability

- *Mutual exclusion*
  As with *Erlang* we let a separate process handle the state of our buffer, requiring messages to go through that process to

[3] Symmetric MultiProcessing

alter that state. As only one message is handled at a time, mutual exclusion is guaranteed. The buffer as implemented in *JoCaml* can be seen in listing 6:

- *Race conditions*
  Due to the similarity of the *Join-Calculus* and the *Actor Model*, the same problems with race conditions encountered in *Erlang*, are present in *JoCaml*.

- *Deadlock*
  Waiting on messages is a blocking operation in *JoCaml*, so deadlock is once again an issue. In *JoCaml* these blocking mechanisms on channels are also used to simulate condition variables, as *JoCaml* doesn't support guards on the patterns selecting messages as *Erlang* does. Listing 6 shows how this works and why it has a tendency to produce messy code.

Listing 6: A buffer process in JoCaml

```
let create_buffer(capacity) =
  def state(x::xs) & get() =
    if (length xs) = capacity
      then state(xs) & reply x to get
    else notfull() & state(xs) & reply x to get
  or state(xs) & put(x) & notfull() =
    if (length xs) < capacity-1
      then notfull() & state(xs@[x]) & reply to
           put
    else state(xs@[x]) & reply to put
  in
    spawn state([]) & notfull();
    {put = put; get = get;}
;;
```

The asynchronous process `notfull` is spawned on creation of the buffer with an empty message, this message is consumed when matched in combination with a message on the put channel. Because the pattern governing the `put` is also joined (with the & operator) to the `notfull` channel, it will block when there is no message on either channel. Both `put` and `get` once again put an empty message on the `notfull` channel when their action results in a not full buffer. Thus the channel simulates a condition variable as seen with the monitor sample, having an empty message for signal and no message for wait. Meanwhile the `state` process is used to keep track of the internal state of the buffer, similarly to the `buffer` process in *Erlang*.

- *Starvation*
  JoCaml uses nondeterministic matching on its patterns, so precedence issues as in *Erlang* are no problem. Unfortunately this also holds for asynchronous messages. *JoCaml* will nondeterministically choose a message from the set of matching messages on an asynchronous channel, without serialization, thus losing sequence. This is the main reason we have used synchronous channels in our sample program, as asynchronous channels would require implementing an *Erlang*-style serializer. The issue of fairness with multiple consumers remains.

- *Nondeterminism*
  In the *JoCaml* sample we use a combination of synchronous and asynchronous channels, with the same problems regarding nondeterminism as in *Erlang*, with the added nondeterministic matching mentioned while discussing starvation.

- *Lack of predictable composability*
  Just as with *Erlang*, protocol issues remain an issue for composability. If *JoCaml* processes are to be composed successfully, it is required to use a protocol governing your messages that is designed with composability in mind.

- *General remarks*
  JoCaml at this point still handles as a rather quirky and unfinished language. The runtime has no support for SMP, so the example given will only use a single core, while we are looking for multicore solutions. A solution exists in using multiple instances of the *JoCaml* runtime and using the distributed features of the language to make use of multiple cores, but this is not really all that practical, especially as distributed computing in *JoCaml* has many issues of its own [33].

  The language is also highly unreadable, with many, seemingly unrelated, overloads of the & -operator, shown in listing 7, and the previously shown clutch required to simulate condition variables, due to lack of guards on the join patterns.

Listing 7: Use of the ampersand in JoCaml

```
(*Join Pattern*)
def state(x::xs) & get() = [..]
(*Message Reply*)
[..] then state(xs) & reply x to get
(*Spawning concurrent processes*)
spawn (cstart(); 0) & (pstart(); 0);;
```

6.3.2 *Performance & Efficiency*

While the *OCaml* basis of *JoCaml* is rather fast, it is also wholly singlethreaded, so with regard to speed and efficiency, it won't be a practical solution for quite some time.

6.4 Software Transactional Memory in Haskell

The sample program for *Software Transactional Memory* will be implemented in GHC Haskell version 6.10.3 [35], a pure functional language with mature support for concurrent transactions.

6.4.1 *Usability*

- *Mutual exclusion*
  Transactions in *Haskell* are both `atomic` and `isolated`. Atomicity guarantees that effects of a transaction become visible only at the end of a transaction block, assuming the transaction is successful. Isolation guarantees that a transaction in operation does not notice the effects of other threads during execution. These allow mutual exclusion to be guaranteed during the execution of transactions. At the commit point of the transaction, the mutated values are only written if the state of the variables has not changed during the transaction. The transaction for the producer can be seen in listing 8:

Listing 8: The producer transaction in Haskell

```
atomically
  ( do {
    c <- readTVar count;
    when (c == cap) retry;
    writeTChan buf n;
    writeTVar count (c+1);
  });
```

It reads the current amount of items in the buffer from the transactional variable, retries when it has reached the maximum capacity and else adds *n* to the transactional channel representing the buffer and increased the count by one.

- *Race conditions*
  Since in *Haskell* STM all mutations of transactional variables are guaranteed to happen in transactions, race conditions disappear as they would in the pure form of STM.

Would you however take a language in which side-effects are not so strictly separated from the rest of the code, especially one where memory can be modified at will, it is easy to once again create a situation in which race conditions may occur.

- *Deadlock*
  Haskell STM does not suffer from deadlock as it is lock-free.
- *Starvation*
  Haskell STM is not wait-free, thus starvation can still occur, achieving fairness in *Haskell*'s implementation of STM even appears to be more difficult than in competing models, as seen in Josef Svenningssons Dining Philosophers programs which exhibit starvation in certain cases [46].
- *Nondeterminism*
  Due to the nesting mechanisms, retrying on clashes and the `orElse` construct, *Haskell* STM is highly nondeterministic.
- *Lack of predictable composability*
  STM transactions in *Haskell* are fully composable, a simple example would be the Dining Philosophers by Svenningsson [46].
- *General remarks*
  Side effects are the bane of a practical implementation of STM, as they generally cannot be reverted e.g. in case of I/O. *Haskell*'s typing system with its monadic constructs allows to solve this in a rather beautiful fashion by making STM transactions their own type with associated compile time constraints. Side effects cannot occur in STM transactions and as STM transactions themselves are side effects as they modify memory, they can only occur in designated blocks. Listing 9 illustrates this:

Listing 9: Side effects in Haskell

```
createBuffer :: Int -> STM Buffer
createBuffer capacity = do
{[STM Actions]}

main :: IO()
main = do
{
  b <- atomically(createBuffer 10);
  [..]
}
```

The first line is a type definition stating that `createBuffer` takes an `Int` and returns a `Buffer` with an `STM` side effect, which must be so as `createBuffer` does `STM`-actions. The function `main` is defined to return nothing, but with `IO` side-effects. `atomically` initiates a transaction, allowing only `STM`-actions and returns an `IO`-action, a more permissive type, describing that side-effects may have occurred.

As most languages do not have a type system as strict as *Haskell*, or even allow direct access to memory, STM does not appear to be a safe solution for the currently popular languages. Syntactically transactions in *Haskell* are quite easy to understand and use, as they essentially mimic the imperative programming style used in single threaded programming in common languages. Unfortunately, it does tend to stand out a bit against the rest of the more functional code generally written in *Haskell*. This is a good thing on the one hand, as imperative sections are clearly denoted, but it does allow you to forget that *Haskell* is foremost a functional language. One could end up with an overuse of `atomic` blocks in the same way that in concurrent java code some people tend to make all methodes `synchronized`.

6.4.2 *Performance & Efficiency*

The GHC-runtime system maps lightweight user thread to kernel threads, with generally one kernel thread per processing unit. The state of each user thread, commonly called a *Haskell thread*, is kept in a thread state object on the heap. The amount of kernel threads, called *worker threads*, is generally equal to the amount of processing units available, but the threads themselves are not fixed to a single unit, allowing them to be moved between processing units.

For each physical processing unit a HEC[4] is kept, containing the mapping to a working thread, a run queue of Haskell threads, a message queue with messages from other HECs and additional data needed for bookkeeping such as the spark pool.

GHC also features a parallel Garbage Collector since version 6.10, however this is still a stop-the-world form of garbage collection and not an independent one, resulting in a massive slowdown when GC occurs [36].

This results in GHC being a fast compiler with decent scaling in the current stable version, and good scaling in the currently experimental version 6.11, with optimizations actively being worked on.

6.5 Runtime Scheduling in C using OpenMP

Our sample bounded buffer for *OpenMP* is implemented in *C*, with `#pragma`-directives specifying the *OpenMP*-constructs. We use the GCC 4.3.3 [47] and the Intel® C++ Compiler Professional Edition 11 [14] as our compilers.

6.5.1 *Usability*

- *Mutual exclusion*
  Mutual exclusion in *OpenMP* is guaranteed by synchronization clauses such as `atomic` and `critical section`. In our sample buffer in listing 10 we specify the critical sections as such:

Listing 10: Mutual exclusion in OpenMP

```
#pragma omp parallel shared([..])
{
  #pragma omp sections
  {
    #pragma omp section //producer
    {
      while(true)
      {
        while (!full)
        #pragma omp critical (buffer)
        {
          [..]
        }
      }
    }
    [..]
  }
}
```

The sections will be parallellized by the compiler, the critical section will be used for mutual exclusion and the shared construct specifies which variables are shared. "(buffer)" is used to specify a name for the critical section, the producer side will use the same name, allowing the compiler to determine that those critical sections need to be mutually excluded.

[4] Haskell Execution Context

- *Race conditions*
  The synchronization `pragma`'s also help against race conditions, but as they must be added manually and it isn't always clear where they should be added, *OpenMP* does little to prevent races from occurring. Listing 11 shows how easily race conditions are created by forgetting a `critical section`.

Listing 11: Incorrect sample with race condition

```
1 int i=0;
2 #pragma omp parallel
  {
4   i++;
  }
```

To get correct behaviour the listing should be modified as in listing 12:

Listing 12: Correct sample without race conditions

```
1 int i=0;
  #pragma omp parallel
3 {
    #pragma omp critical
5   i++;
  }
```

- *Deadlock*
  Deadlock is still a likely occurrence in *OpenMP*, especially when using nested critical sections or locks. A sample is given in listing 13:

Listing 13: Deadlock in OpenMP

```
1  #pragma omp parallel
2  private(i)
   {
4    int i;
     i = omp_get_thread_num();
6    if (i == 0) goto Master;
     #pragma omp barrier
8    Master:
       #pragma omp single
10     write(*,*) "done"
   }
```

The deadlock here occurs due to the `goto`-statement allowing threads to arrive at different barriers, thus causing problems with regard to sequence of taking locks.

- *Starvation*
  OpenMP gives no guarantee of fairness when entering a critical section, however, it does guarantee forward progress. While this allows threads to be starved when other threads get repeated access to the critical section, eventually a thread will acquire access [11].

- *Nondeterminism*
  As *OpenMP* builds upon the lock-based model, nondeterminism is high. The timeslicing is OS-dependent and because of the runtime scheduler it is even less clear what will be run at which time.

- *Lack of predictable composability*
  The annotations combined with the runtime scheduler allow for better composition, as *OpenMP* can determine a solution for composition at compile and run time, optimizing cases a programmer could miss. However, as *OpenMP* still allows you to take locks in the traditional fashion with `omp_init_lock`, `omp_set_lock` and `omp_unset_lock`, it is still easily possible to write code that does not compose correctly.

- *General remarks*
  The buffer sample used displays radically different behaviour depending on the compiler used, even deadlocking in certain instances.

  Because there are no condition variables in *OpenMP*, as it is more a system for large scale autoparallelization and as such it has limited work sharing constructs, the current solution for the bounded buffer used busy waiting instead of waiting on conditions.

6.5.2 Performance & Efficiency

OpenMP defaults to using kernel threads, with the amount equal to the physical processing units available. It supports either static scheduling or runtime scheduling, determined by a compiler flag, as in certain cases the ability to allow for static scheduling allows performance benefits.

The performance is mostly dependent on the compiler and library used, but when using a mature C-compiler such as the Intel® C++ Compiler, it will generally surpass the alternatives listed, especially in large parallel systems, as *OpenMP* tends to scale quite well.

In cases of insufficient granularity provided by the `#pragma`-directives, manual locking is still an option.

7. CONCLUSIONS

Although there appear to be no drop in replacements for the lock-based model of concurrency, the alternatives that exist are certainly viable. While the models themselves tell us something about the underlying philosophies, the implementations are what really matters. It seems impossible to linearly rank the different implementations, as none proved to the strictly better that the others, so we shall give some general conclusions about each model and when they can or should be used.

The *Actor Model* as implemented in *Erlang* is easy to use and reason about in simple cases, as asynchronous messaging can essentially be seen as events happening on processes that should be handled. In larger programs however, a lot of attention should be paid to developing robust messaging protocols, as protocol errors seem to be the most frequently occurring. Performance wise, Erlang is a somewhat slow language that scales quite well on smaller SMP systems, but as of yet not on larger clusters.

The *Join-Calculus* as a model seemed rather promising, but it suffers in implementation in *JoCaml*. The lack of guards on join-patterns, requiring condition variable-like hacks, and the odd syntax do not help usability wise and performance suffers due to lack of support for SMP and unfinished support for distribution.

Software Transactional Memory in *Haskell* allows one to program as writing serial code while writing concurrent code, which allows for large ease of use, however, serial code does seem somewhat out of place in a pure functional language. The monadic approach in *Haskell* however does raise the question of the viability of implementing STM-like models in currently popular languages, as they have no direct means to prevent you from causing side-effects in transactions. Performance wise GHC is fast and getting faster, however intrinsic problems in *Software Transactional Memory*, such as processes starving due to massive retries, do hamper performance.

OpenMP appears to be simple to use, as one only has to annotate one's existing code to gain the benefits of parallel execution, however, the annotations themselves still require complex reasoning about concurrent situations, essentially giving you all the problems of the *lock-based* model. The ability to revert to this model

is both a blessing and a curse, as it allows both highly granular optimizations and interesting ways to shoot oneself in the foot by interfering with the *OpenMP* scheduler. The different implementations of the library and the different results gained by compiling with different compilers are an unfortunate side effect. The benefit in raw performance is there, especially for large systems.

On existing projects entrenchment and maturity will keep the *lock based concurrency* the model to use for general applications for the foreseeable future. Models that build on this, such as *Runtime Scheduling*, will be the first step in overcoming this and will almost always be an improvement.

On newer projects, especially when you are familiar with the implementation languages, *Software Transactional Memory* as implemented in *Haskell* and the *Actor Model* as implemented in *Erlang* are definitely worth using, as they prevent much of the problems inherent with the lock based model.

REFERENCES

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[3] Russell Atkinson and Carl Hewitt. Synchronization in actor systems. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 267–280, New York, NY, USA, 1977. ACM.

[4] M. Ben-Ari. *Principles of concurrent and distributed programming, second edition*. Addison-Wesley, 2006.

[5] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. Technical report, Cambridge, MA, USA, 1996.

[7] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, 1998.

[8] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[9] Richard Carlsson and Hakan Millroth. Towards a deadlock analysis for erlang programs, 1997. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.8054`.

[10] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[11] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[12] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, 1984.

[13] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[14] Intel Corporation. Intel® compilers. `http://software.intel.com/en-us/intel-compilers/`, 2009.

[15] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[16] Robert Ennals and Robert Ennals. Software transactional memory should not be obstruction-free. Technical report, Intel Research Cambridge, 2006.

[17] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385, New York, NY, USA, 1996. ACM.

[18] Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, pages 268–332, London, UK, 2002. Springer-Verlag.

[19] Brian Goetz. Java theory and practice: Urban performance legends. `http://www.ibm.com/developerworks/java/library/j-jtp04223.html`, 2003.

[20] Sun Information Products group. About openmp. `http://openmp.org/wp/about-openmp/`, 2008.

[21] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: a benchmark for software transactional memory. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 315–324, New York, NY, USA, 2007. ACM.

[22] Philipp Haller. Scala actors: A short tutorial. `http://www.scala-lang.org/node/242`, 2008.

[23] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

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

[25] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[26] Frank Huch. Verification of erlang programs using abstract interpretation and model checking. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 261–272, New York, NY, USA, 1999. ACM.

[27] James Iry. Erlang style actors are all about shared state. `http://james-iry.blogspot.com/search/label/erlang`, 2009.

[28] G Stewart Itzstein and Mark Jasiunas. On implementing high level concurrency in java. In *In Proceedings of the Eighth Asia-Pacific Computer Systems Architecture Conference*, pages 151–165. Springer, 2003.

[29] Leslie Lamport. Solved problems, unsolved problems and non-problems in concurrency. *SIGOPS Oper. Syst. Rev.*, 19(4):34–44, 1985.

[30] Jim Larson. Erlang for concurrent programming. *Queue*, 6(5):18–23, 2008.

[31] Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan 2006. The published version of this paper is in IEEE Computer 39(5):33-42, May 2006.

[32] Yigong Liu. Boost.join. `http://freshmeat.net/projects/boostjoin/`, 2008.

[33] Louis Mandel and Luc Maranget. The jocaml language user manual on distributed programming. `http://jocaml.inria.fr/manual/distributed.html`, 2008.

[34] Louis Mandel and Luc Maranget. The jocaml system. `http://jocaml.inria.fr/manual/index.html`, 2008.

[35] Simon Marlow. The glasgow haskell compiler. `http://haskell.org/ghc/`, 2009.

[36] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. Submitted to ICFP'09, March 2009.

[37] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[38] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[39] C. Perfumo, N. Sönmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, and Valero M. The limits of software transactional memory (stm): dissecting haskell stm applications on a many-core environment. In *Proceedings of the 2008 conference on Computing frontiers (CF '08)*. ACM, May 2008.

[40] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, 1977.

[41] The Moscova Project. The join calculus language. `http://moscova.inria.fr/join/index.shtml`, 2007.

[42] K. Sagonas, M. Pettersson, R. Carlsson, P. Gustafsson, and T. Lindahl. All you wanted to know about the hipe compiler (but might have been afraid to ask), 2003.

[43] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[44] Eugene W. Stark. Semaphore primitives and starvation-free mutual exclusion. *J. ACM*, 29(4):1049–1072, 1982.

[45] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[46] Josef Svenningsson. Some examples of software transactional memory in haskell. `http://computationalthoughts.blogspot.com/2008/03/some-examples-of-software-transactional.html`, 2008.

[47] The GCC Team. Gcc, the gnu compiler collection. `http://gcc.gnu.org/`, 2009.

[48] John A. Trono. A new exercise in concurrency. *SIGCSE Bull.*, 26(3):8–10, 1994.

[49] T. Willhalm and N. Popovici. Putting intel®threading building blocks to work. In *Proceedings of the 1st international workshop on Multicore software engineering (IWMSE '08)*. ACM, May 2008.