Programming a Multicore Architecture without Coherency and Atomic Operations

Jochem H. Rutgers Marco J.G. Bekooij Gerard J.M. Smit

University of Twente, Department of EEMCS
P.O. Box 217, 7500 AE Enschede, The Netherlands
{j.h.rutgers, m.j.g.bekooij, g.j.m.smit}@utwente.nl

Abstract

It is hard to reason about the state of a multicore system-on-chip, because operations on memory need multiple cycles to complete, since cores communicate via an interconnect like a network-on-chip. To simplify programming, atomicity is required, by means of atomic read-modify-write (RMW) operations, a strong memory model, and hardware cache coherency. As a result, multicore architectures are very complex, but this stems from the fact that they are designed with an imperative programming paradigm in mind, i.e. based on threads that communicate via shared memory.

In this paper, we show the impact on a multicore architecture, when the programming paradigm is changed and a λ-calculus-based (functional) language is used instead. Ordering requirements of memory operations are more relaxed and synchronization is simplified, because λ-calculus does not have a notion of state or memory, and therefore does not impose ordering requirements on the platform. We implemented a functional language for multicores with a weak memory model, without the need of hardware cache coherency, any atomic RMW operation, or mutex—the execution is *atomic-free*. Experiments show that even on a system with (transparently applied) software cache coherency, execution scales properly up to 32 cores. This shows that concurrent hardware complexity can be reduced by making different choices in the software layers on top.

Categories and Subject Descriptors C.1.3 [*Processor Architectures*]: Other Architecture Styles; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Design, Performance, Languages

Keywords memory model, cache coherency, distributed shared memory, embedded system, functional language

1. Introduction

Multicore processors are generally accepted as computing platform. From a peak-performance point of view, it seems logical to keep on adding more cores on a chip in order to get more performance [1]. However, scaling a few-core symmetric multiprocessing architecture to a many-core one introduces several problems in the hardware architecture. Where two cores can share one bus, larger systems often have complex interconnection structures like a network-on-chip, which increase the latency of the communication between cores. This complicates operations that need atomic global communication, where the result of such an operation becomes visible to all cores, without any (observable) intermediate state during the state transition. Examples of such global communication are: a hardware cache coherency protocol that is getting exclusive access to perform a write [1, 2]; executing an atomic read-modify-write (RMW) operation, like a compare-and-swap; and even a single write operation in a system with a memory model that requires a total store order [3].

Alternatives that avoid atomics, and are therefore easier to realize in hardware, are software cache coherency [2], or not to use cache coherency at all, like the 48-core Intel SCC [4]. Additionally, when a weak memory model is used, a total order of memory operations might not have to be guaranteed. However, these alternatives complicate programming. When state changes are not instant anymore, and need some time to complete, the transient state is unpredictable, but still observable in a multicore environment. The hardware often exposes these issues to the programmer, which makes reasoning about correct program behavior very hard [5].

The problems mentioned above are all related to memory consistency and synchronization, or concurrency in general. A widely-used concurrent programming paradigm to harness the power of a parallel machine is threading in combination with shared memory. However, Lee [6] argues that threads (in combination with an imperative language like C) induce non-determinism, which all should be pruned away by the programmer. Having a strong memory model and efficient synchronization makes this task a bit easier, but also make the hardware more complex and less scalable, which leads to the problems above. Hence, the choice of a programming paradigm strongly influences the design choices regarding the hardware platform. We argue that the synergy of the programming model and the platform should be exploited, and we want to demonstrate that the atomic communication issues mentioned above can be solved at a higher level by changing the programming paradigm.

In this paper, we show that the requirements for a multicore architecture relax, when assuming that it is programmed using a functional language. More specifically:

1. We show that concurrent execution can be achieved without locks and atomic RMW operations, even on hardware with a weak memory model, based on properties of the programming paradigm. Hence, the execution is *atomic-free*; it does not rely on any sequence of operations that should be observed by or communicated to other processors atomically, in either hardware or software. This opens the possibility to reduce

PMAM'14, February 15-19 2014, Orlando, FL, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2655-1/14/02...$15.00.
http://dx.doi.org/10.1145/2560683.2560697

hardware complexity, and therefore makes the hardware more scalable. We acknowledge that avoiding *all* atomics is a very strong requirement, and that practical systems might benefit by allowing some of them anyway. However, we show that it is possible to do so, and still provide a proper programming interface.

2. We show that carefully introducing data races in the run-time system does *not* harm the deterministic behavior of the application, which is in contrast to data races in the C11 standard.
3. We derive rules for a weak memory model, and show the relation to different memory models and (transparently applied) software cache coherency.
4. Experiments show the feasibility of the approach.

This can be achieved, because of the nature of the λ-calculus, which is the mathematical basis of the functional programming paradigm. Its distinctive properties include that it does not have the notion of state or memory, which eases dealing with weak memory models. Moreover, functional programs naturally allow concurrency, because all dependencies between calculations are explicitly defined. Furthermore, since a functional language is single-assignment, calculating the same expression twice gives the same result, which allows simplification of synchronizing concurrent calculations.

1.1 Basic Idea

The basic idea of this approach is exemplified as follows. Consider the following pseudo-code:

```
x = foo();
y = bar() + x;
```

If this code snippet was C, the assignments of x and y should be done in the specified order, otherwise the initial value of x is used for addition instead. When both lines are calculated by different threads, the computation of y by one thread should be stalled until it is guaranteed that the other thread finished computing x.

In a functional language, variables are single-assignment, so '=' means *definition* instead. One can imagine that the thread calculating y checks whether x has been computed yet, and if not, it can do it by itself. Hence, a data race exists in the calculation and assignment of x. However, even if x is evaluated twice because of this data race, the result is the same.

Allowing this data race can be used for optimizations, without influencing the outcome of the program. Threads might decide to compute variables repeatedly to prevent fetching it from shared memory and consuming precious memory bandwidth. Additionally, distributing work among worker threads without proper synchronization might also (safely) result in duplicates. Moreover, communication of results to other cores can be postponed when that seems to be beneficial for cache coherency protocols, for example.

However, there is no free lunch. In contrast to C, it is not obvious to decide whether a variable is still in use, as the source code does not define when a variable is not used anymore. Keeping administration at run time is possible, but data races might complicate such analysis. So, there is a balance between allowing races and arbitrary ordering of memory operations during evaluation, and guarantees about the memory state for garbage analysis.

The rest of the paper works out this basic idea and is structured as follows. Related work is discussed in Section 2. To evaluate the principles of λ-calculus in a many-core environment, we implemented an untyped functional language. Section 3 elaborates on the details. As suggested above, orderings of reads and writes can be less strict. Section 4 discusses the impact of this on the memory model, and presents a simple though efficient software cache coherency protocol as a proof-of-concept. Section 5 presents the results of running applications of the NoFib Benchmark Suite on a 24-core Intel machine and a non-cache-coherent 32-core MicroBlaze system on FPGA. Section 6 concludes the paper.

2. Related Work

Many functional languages exist, and they handle concurrency (and the related problems) differently. Clo*j*ure [7] runs in the Java VM and assumes worker threads on top of a shared-memory machine. SAC is based on a fork-join approach [8]. Haskell supports different flavors of parallelism, based on the Glasgow Haskell Compiler (GHC): annotations and implicit concurrency [9], explicit threads and channels [10], and data parallelism [11]. All implementations assume running on an SMP machine, with a POSIX-like OS, which implies having a strong memory model and threads. Ports of Haskell to other architectures include House [12] and GHC's port to ARM, but they do not support multiple cores. This paper focuses on the fundamental requirements of executing a parallel functional program, instead of assuming a common architecture. To the best of our knowledge, no work focuses on the direct relation between these languages and an underlying hardware architecture.

Other parallel functional languages are based on message passing, like Erlang [13], Eden [14] and Multi-MLton [15]. These languages can be ported to many architectures, because the message-passing abstraction hides issues related to memory consistency, and the model fits nicely to networks of computers. The same holds for stream processing applications that are implemented using message passing. However, sending and receiving messages has overhead by (unnecessary) data duplication, and it enforces a specific form of synchronization. As shared memory is more generic [16], and we focus on concurrency within a single system-on-chip, message passing is not considered in this paper.

In a different direction, the functional programming paradigm can also influence processor design instead of the system architecture. PilGRIM [17] is an example of a specialized processor for lazy languages. The authors propose a multicore system as future work. However, it likely encounters the same memory consistency issues as any other multicore system, as the memory layout of expressions during execution is similar to the one discussed in Section 3.3.

Although not specifically for functional languages, Bhattacharjee et al. [18] measured the synchronization primitive overhead of the parallelization libraries OpenMP and Intel's TBB. Even though the authors propose optimizations to improve the measured overhead of respectively 47% and 80% of the benchmark run time, they conclude that the overhead remains high at higher core counts. In contrast, we eliminate the need for such synchronization primitives, by choosing a different programming paradigm than the one of C.

Avoiding locks and atomic instructions is also proposed by Tithi et al. [19] for breadth-first search algorithms. These operations are recognized as costly, and their proposed solution outperforms state-of-the-art algorithms. Nasre et al. [20] also conclude that RMW operations are costly and discuss transformations of graph algorithms to eliminate them, specifically targeting GPUs. These techniques modify the algorithm, where we avoid atomic operations in general at the level of the programming paradigm. Optimizing the algorithm itself is beneficial, and it is an orthogonal technique to the modifications to the platform, as discussed in this paper.

On larger scale systems like cluster computers, MapReduce [21] is a popular approach to program for concurrency. In this model, a function is concurrently applied to every element of a large dataset (*map*), and the individual results are combined into a smaller dataset, like the sum of the inputs (*reduce*). Because both phases do not modify the dataset, they can be considered side-effect free, which in turn can tolerate processing node failures by reissuing work. Although MapReduce assumes 'embarrassingly' parallel applications and large datasets on disks, the benefits also apply to functional languages in a smaller scale, like a single multicore system. However, functional languages are more generic, as they do not assume such a specific form of parallelism in the application.

3. Shift in Paradigm: λ-Calculus and Its Implementation

To understand the execution and memory related issues of programs written in a functional language, we (informally) explain the fundamentals of such a language. Afterwards, the implementation of our functional language is discussed, which closely follows these fundamentals.

3.1 Background on λ-Calculus

Functional languages are based on their counter part in mathematical logic, λ-calculus. This formal system defines expressions or *λ-terms* as

$$M ::= c \mid x \mid M\ M \mid \lambda x.M$$

where c can be any constant or primitive function, x is a variable that binds a name to another λ-term, $M\ M$ is an application of the second λ-term to the first one, and $\lambda x.M$ is a function that takes one argument and binds it to all occurrences of x in M. Prefix notation is used for function application. For example, $(\lambda x.((+\ 2)\ x))$, or just written $\lambda x. +\ 2\ x$, is a function that takes one argument and adds 2 to it (although the $+$ operator does not exist in λ-calculus, but assume that its behavior is defined).

In this simple example, $+$ is (assumed to be) a function that takes two arguments. If only one argument is applied to it, like $(+\,2)$, the result is still a function, but now requires one argument—supplying fewer arguments than the function requires is *partial application*. Functions can also be used as arguments. Functions that can take and/or return functions are *higher-order functions*. An example is function composition, $f \circ g$, which is defined as $C := \lambda f.\lambda g.\lambda x.f\ (g\ x)$. When both f and g are applied to C, the result is a function that still requires one argument.

Next, the only rule of computation is called *β-reduction*, which substitutes a formal argument by an actual one. So, $(\lambda x. +\ x\ 2)\ 3$ is reduced to $+\ 3\ 2$, which then can be computed as 5.

The order in which expressions should be reduced is not defined. For example, $(\lambda x.f\ x)\ ((\lambda y.g\ y)\ 7)$ can first be reduced to either $(\lambda x.f\ x)\ (g\ 7)$ or $f\ ((\lambda y.g\ y)\ 7)$. However, based on the Church-Rosser Theorem [22], the fully reduced result is always the same—in this case $f\ \ (g\ \ 7)$. Evaluating a function is *side-effect free*; it only computes a result, but does not change the system in any other way. Therefore, reducing a term can be postponed until its value is required, which is exactly what a *lazy* functional language does.

A more program-like example is the following definition of the volume of a cylinder with radius 2 and length 5:

$$\begin{aligned} \mathrm{main} &= \mathrm{cylinder}\ 2\ 5 \\ \mathrm{cylinder} &= \lambda r. \times\ (\times\ \pi\ (\mathrm{sqr}\ r)) \\ \mathrm{sqr} &= \lambda x. \times\ x\ x \end{aligned}$$

When we repeatedly reduce main, the result is computed:

$$\begin{array}{ll} \mathrm{main} & \rightarrow_{\text{substitute}} \\ \mathrm{cylinder}\ 2\ 5 & \rightarrow_{\text{substitute}} \\ (\lambda r. \times\ (\times\ \pi\ (\mathrm{sqr}\ r)))\ 2\ 5 & \rightarrow_{\beta} \\ (\times\ (\times\ \pi\ (\mathrm{sqr}\ 2)))\ 5 & \rightarrow_{\text{substitute}} \\ (\times\ (\times\ \pi\ ((\lambda x. \times\ x\ x)\ 2)))\ 5 & \rightarrow_{\beta} \\ (\times\ (\times\ \pi\ (\times\ 2\ 2)))\ 5 & \rightarrow \\ (\times\ (\times\ \pi\ 4))\ 5 & \rightarrow \\ 62.83... & \end{array}$$

```
// library
class Term {
  Term& operator()(int c){
    return *new Application(*this,*new Constant(c));}
  Term& operator()(Term& t){
    return *new Application(*this,t);}
  // beta-reduce this term
  virtual Term& Reduce(){return *this;}
  virtual Term& Apply(Term& args...);
};
class Constant : public Term {
  Constant(int i); };
class Application : public Term {
  Application(Term& func,Term& arg);
  virtual Term& Reduce(){return func.Apply(arg);}
  virtual Term& Apply(Term& args...){
    return func.Apply(arg,args...);}
};
class Function : public Term {
  Function(Term& (*func)(...));
  // omitted: handling superfluous arguments
  virtual Term& Apply(Term& args...){
    return func(args...);}
};

// standard library with many functions, including:
Function mult;
```

Listing 1. Simplified implementation of LambdaC++

We implemented a simple functional language that closely follows the definition of λ-calculus and the β-reduction rule, which is discussed next.

3.2 Our Simple Functional Language: LambdaC++

Based on the definition of λ-calculus, it does not fundamentally require hardware features such as a strong memory model and fully deterministic execution. As a proof-of-concept to show that it is possible to realize an atomic-free execution of a concurrent program, we implemented an untyped functional language[1]. In fact, the language is just C++, where λ-terms are represented by functors—classes that overload the `()`-operator, such that the syntax resembles λ-calculus somewhat and functions can be used as function arguments. In this paper, we will refer to this language and its implementation as *LambdaC++*.

We will discuss the aspects of the implementation where usually atomics are involved. Notably, the focus is on data races during β-reductions, and the distribution of work via a lossy work queue.

3.2.1 General Setup

A simplified implementation of the (program-independent) C++ classes is shown in Listing 1. Among many other details, handling of multiple arguments and partial function application are left out for simplicity. The example of the previous section can be implemented as depicted by Listing 2. Because `g++` is used as the compiler, optimizations are only applied at the C++ level; `g++` is oblivious to the functional behavior and properties of the program. Although adapting GHC and using Haskell would be better and possible, modifying the fundamentals of such a large project is practically not feasible for us at the moment.

The implementation supports integers, (complex) doubles, and arbitrary large numbers via the GNU MP library, although the language is in principle untyped and the compiler does not check these types. A list is defined as a chain of Church pairs, which is a function $\lambda r.\lambda l.\lambda c.c\ r\ l$ assuming that the function c returns either r

[1] The implementation is available under the GPLv3 license at `https://sites.google.com/site/jochemrutgers/lambdacpp`.

```
Term& sqr_func(Term& x){
  return mult (x) (x);}
Function sqr(sqr_func);

Term& cylinder_func(Term& r){
  return mult (mult (pi) (sqr (r))); }
Function cylinder(cylinder_func);

Term& main_func(){
  return cylinder (2) (5); }
Function main(main_func);
```

Listing 2. Example program in LambdaC++

or l, depending whether the right or left term is requested. Because C++03 does not allow anonymous functions, the λ-expression of the form $\lambda x.M$ should be lambda lifted [23]; it can only be defined as a named function, like sqr and cylinder in Listing 2, and not occur somewhere inline.

3.2.2 Worker Threads

Concurrency is exploited by running one worker thread per core, which all concurrently reduce parts of the program. Each thread has its own heap that contains λ-terms. Parallelism is introduced by the par function, which is very similar to the one of Parallel Haskell. This function pushes one of its arguments on a work stack, allowing other worker threads to pick it up and start to eagerly reduce the term. This introduces the notion of local and shared data: all data is local, until it is applied to par. Then, the terms are made globally visible. The implementation ensures that local terms can refer to both local and shared ones, but shared terms only refer to other shared terms [24]. So, terms are either global or local and are always owned by the worker thread that owns the heap it resides in. To decide when a worker can free terms in its heap, garbage collection (GC) is required. A worker can be in a few different phases: idle when out of work, running β-reductions, and doing GC. Unlike GHC, when evaluation of a term blocks (for example on a term that is currently evaluated by another worker), the worker thread just blocks; no context switching has been implemented between evaluation of multiple (unrelated) terms.

Everything that happens at run-time, such as doing β-reductions, handling of distribution of work among workers, allocation and garbage collection of memory, are part of the run-time system (RTS).

3.2.3 Local vs. Global Garbage Collection

Multiple approaches to garbage collection exist. As the presented concepts in this paper are independent of the chosen approach, we only briefly discuss a high-level overview of the approach we use.

We chose to use a mark-sweep approach [25]. The algorithm works according to the following steps: 1) it marks all terms on its heap as dead; then 2) it marks all terms that are pointed to from the program stack as alive; next 3) it follows pointers from living terms to other terms, until no new living term is found; and finally 4) all dead terms are freed. To find the root of the computation, a shadow stack [26] is used to track all active references to terms. There are two flavors of GC:

- Local: Only local terms are cleaned from the heap. This can be done independent of other workers, because it is guaranteed that all local terms are not used by other workers. All encountered shared, i.e. global, terms are assumed to be alive. Because only locally accessible terms are processed during local GC, memory consistency is irrelevant; no other worker reads or writes these terms.

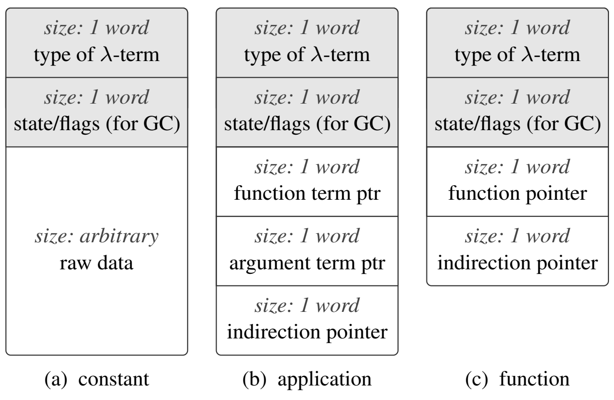


Figure 1. Memory layout of λ-terms

- Global: Local and shared terms are cleaned from all heaps. This can only be done in a stop-the-world fashion, where all workers stop the current evaluation and participate in a GC run. The synchronization between workers can be done by a (Pthread) barrier. Although such a barrier could be relatively expensive, it does not influence the performance much, as this is a relatively rare operation—β-reductions are done orders of magnitude more often. For example, a shared-memory polling-based algorithm like the bakery lock [27] suffices. As this paper focuses on concurrency issues during evaluation, a discussion about the internals or optimization of the GC is beyond the scope of this paper. Moreover, as the GC is written in ‘normal’ C++, it uses weak memory models in a general fashion, which has been covered in earlier work [28].

The local GC is invoked when the currently allocated heap memory is exhausted. When not enough garbage is collected, more memory is requested from the OS. Global collection is invoked every second, but never in the midst of an arbitrary function; the RTS can only switch to GC when it is idle, or a new term has to be created and new memory is allocated. In contrast to interrupts, which can arrive at any time, the execution of the program is therefore always in a known state.

3.3 The Atomic-Free Core: Into Evaluation with Data Races

As discussed Section 3.1, computation in λ-calculus is done by repeatedly doing β-reduction on a term, until it results in a constant, or it is a partially applied function. Since a β-reduction is side-effect free, it is safe to allow some non-determinism. First, we explain how the data structures and reduction steps are organized.

The RTS replaces a reduced term by the reduction result, instead of that it modifies the term. Because the result does not have to be of the same size, it is easier to allocate new memory for the result, and set an indirection pointer to that result. Therefore, the contents of a term never changes, it can only get superseded when its indirection pointer is set. Given the information required for λ-terms in general and this approach to do β-reductions, we derive a generic memory layout, which is depicted in Figure 1. Every term has in common that it contains its type (the vpointer, in C++ lingo). Next, administrative fields are added, depending on the GC approach. A constant then only has to contain the raw data. An application requires a pointer to the argument that is applied and the function that it is applied to—when a function requires multiple arguments, a chain of applications is used. A (named) function, like sqr, needs to store the function pointer, e.g., to `sqr_func`, to call upon computation. Constants cannot be reduced further, but functions without arguments and applications need to store the

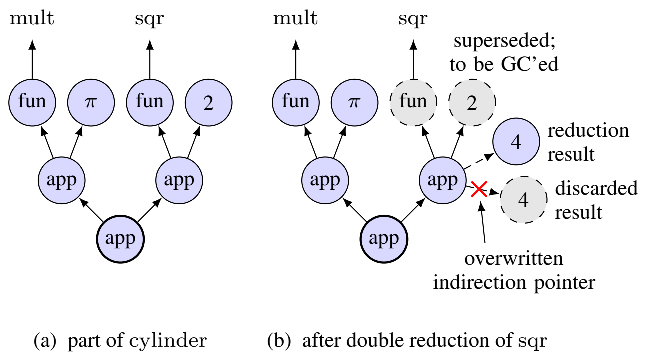


Figure 2. Steps in computation

indirection pointer to the reduction result. In the implementation, when the indirection pointer is set, the function and argument that are pointed to, are not considered to be required anymore and can be garbage collected eventually. Therefore, all contents are constant after initialization of the term, except for the indirection pointer.

To clarify the data structures, Figure 2(a) exemplifies a part of the total graph of cylinder of the example of Listing 2. When the application of sqr and 2 is reduced, the result is 4 and the application term is indirected to this result. Figure 2(b) shows the graph when two workers have reduced the same term at the same time. Both workers can update the indirection pointer, which results in a data race. However, it does not matter how others observe this update; the result is the same either way. During GC, the race is reconciled and one result is properly discarded.

The fact that the race condition can safely be ignored and that setting the result does not require locks, is a great potential for relaxing constraints on the memory model. Section 4 will define what is required to allow these races.

3.4 Work Distribution via Work Queue with Data Races

As mentioned before, all workers get their work via a work queue. A thread-safe queue usually requires mutual exclusive protection to guarantee proper handling of the elements in the queue. However, we will show that the evaluation model allows the queue to be *lossy*. As a result, we can allow data races in the queue when pushing and popping work.

The work queue is populated using par. We now take a closer look how the par function works. Consider the following definition:

$$\begin{aligned}
\text{par} &= \lambda x.\lambda y.y && \text{, additionally, } x \text{ will be pushed on a queue;}\\
\text{pseq} &= \lambda x.\lambda y.y && \text{, additionally, } x \text{ will be fully reduced first;}\\
\text{fib} &= \lambda n.1 && \text{, when } n \leq 2;\\
\text{fib} &= \lambda n.+\ (\text{fib}\ (-\ n\ 1))\ (\text{fib}\ (-\ n\ 2)) && \text{, otherwise;}\\
\text{pfib} &= \lambda n.\text{pseq}\ (\text{par}\ a\ b)\ (+\ a\ b)\\
& && \text{, where } a = \text{fib}\ (-\ n\ 1)\text{, and } b = \text{fib}\ (-\ n\ 2).
\end{aligned}$$

par is used as an annotation to indicate that the value of its argument is expected to be needed in the future and that the computation might take some time. On the other hand, pseq breaks the normal lazy reduction order by forcing to compute the first argument before continuing with the rest. So, the functions par and pseq do not influence the outcome of the program, but are just hints how the program might be executed faster.

fib calculates (inefficiently) the Fibonacci number of a given index in the sequence. pfib does the same, but first puts a on a work queue, then calculates b, and adds them afterwards. Both a and b are variables, and each of them is bound to an application that will eventually indirect to a constant. When the addition of a and b is calculated, the worker has to follow the indirection pointers, until it encounters this constant term. However, par does not guarantee that the computation of a will be finished before the addition. If a was not computed yet at the time of the addition, it will be computed when the value is needed.

As par hints that its argument requires a significant amount of work to compute, it is probably wise to make sure that this computation is done only once, in contrast to the example of Figure 2. To this end, when one worker evaluates the term and another one requires it meanwhile, the second worker should *stall* until the first one has finished computation. If the second worker would also start computing the term, performance is wasted. So, par does the following:

1. Make sure that the term is globally visible, by duplicating it as a global term. Such a global term can reside in the same heap as the local term does, but the C++ class just handles accesses to it differently.
2. Add a *black hole* to prevent double work, by marking a term 'being under evaluation', such that other workers can wait for the result. In the implementation, a black hole is a subclass of `Term`, which eagerly reduces the term it points to and sets its indirection pointer to the result afterwards. When another worker tries to reduce the black hole, it stalls until the indirection pointer is written.
3. Put a reference to the black hole (and therefore the duplicated term) on a work queue.
4. Set the local term's indirection pointer to the newly created (black hole that protects the) equivalent global term.

GHC implements such a queue as a lock-free (work-stealing) FIFO queue. The implementation does not lock a mutex, which otherwise might prevent other threads to progress when the thread that locks it, is context-switched or blocks on another shared resource, for example. However, a lock-free data structure is based on atomic read-modify-write operations, such as a compare-and-swap [29]. These operations are hard to implement in hardware and, more importantly, not required for our queue.

We chose to design this queue as a *lossy stack*. The rationale behind the choice for a stack instead of a FIFO queue, is that newly pushed work onto the stack is more relevant to start computing than older terms, as these older terms are more likely to be computed by the thread that pushed it. The stack can be lossy, because it is allowed that race conditions prevent terms from being pushed at all, and popped terms might be popped twice at the same time. In the former case, the thread that pushed the work will compute the term by itself when required, in the latter case, the black hole will prevent doing the work twice.

Here is a trade-off between allowing incidental losses/duplicates over using locks or RMW operations. In systems with hardware support for the latter, using them can be beneficial. However, we show that the lossy stack allows avoiding atomics, but still guarantees correct program behavior.

The implementation of the lossy stack is shown in Listing 3. `fence()` is a full fence, i.e. memory barrier, equivalent to `gcc`'s `__sync_synchronize()`, such that all preceding reads and writes should be completed and visible for other threads before any successive ones. `flush()` should make sure that any write should be made visible to other threads, which can be used for software cache coherency.

During global GC, the contents of the stack are also used as the roots of computation. Although there are many race conditions in the implementation of Listing 3, these are only relevant during evaluation of the program. It is safe to walk over the stack during

```
class LossyStack {
  int volatile m_top;
  Term* volatile
    m_queue[SIZE];
public:
  LossyStack() : m_top() {}
  void push(Term* term){
    int top=m_top;
    if(top<SIZE){
      m_queue[top]=term;
      flush(m_queue[top]);
      fence();
      m_top=top+1;
      flush(m_top);
    }
  }
  Term* pop(){
    int top=m_top;
    fence();
    if(--top>=0){
      Term* res=
        m_queue[top];
      m_top=top;
      flush(m_top);
      return res;
    }else
      return NULL;
  }
};
```

Listing 3. Lossy stack

global GC, because no worker modifies the stack at that time. Race conditions during evaluation are addressed in the next section.

4. Impact of λ-Calculus on Memory Consistency and Synchronization

The previous section identified two possibilities to allow data races: in the work queue, and during β-reductions of the program. For the former, Section 3.4 presented an implementation. The latter is discussed in this section.

When a functional program is executed concurrently, multiple worker threads reduce terms at the same time and might even reduce the same term simultaneously. Section 3.3 showed that λ-terms are constant during their lifetime, except when the indirection pointer is set after reduction and the term becomes superseded. Based on this sequence, we can derive rules how the memory should behave such that races are allowed, but the program's result is deterministic. This section relates the rules imposed by the programming paradigm to the *memory model* of the platform. First, we briefly discuss memory models in general. Then, we focus on operations on λ-terms, and then make the translation to operations on memory locations and existing memory models.

4.1 Memory Models

A memory (consistency) model prescribes the *guarantees* that processes, i.e. worker threads, can use to deduct conclusions about the memory state, based on the state changes they *observe*. These guarantees involve the *order* in which effects of operations are observed, such as whether it is guaranteed that different workers see changes to different memory locations in the same order. To this extend, an operation is said to be *completed* when every worker will observe the result of that operation. Many different memory models exist and are usually compared based on how strong the models are. In this context, *strong* means that the model defines many guarantees and it is easy to reason about state, and a *weak* model offers less guarantees, but is therefore easier to implement in hardware or is more scalable.

An example of a popular strong model is Sequential Consistency, which is a model that is convenient to reason about; all operations on the memory occur and complete instantly in a specific sequence and all processes agree on that sequence. Because this is hard to realize in hardware, hardware designers rather implement a weaker model, but such a model offers fewer guarantees. Among many other models, the x86 architecture implements a model that ensures that all writes are in total order [30], but earlier executed writes may be observed after specific later executed reads. So, the architecture or the hardware cache coherency protocol has to make sure that some form of synchronization is done at every write to ensure such a total order. A very weak model is Entry Consistency (EC) [31], which divides all reads and writes per shared object—a region in shared memory—in sections with either exclusive or shared access. Shared accesses are read-only and can be done concurrent, where exclusive accesses cannot be concurrently executed to other accesses (not even read-only) of the same memory location. Hence, synchronization is required when any form of access is requested. Finally, Slow Consistency defines only an order per memory location, per processor. The interleaving of these orders is undefined and different processors can even disagree about it, so there is no synchronization required by the hardware. This might sound obscure, but every processor already implements at least Slow Consistency, otherwise any causality of single-threaded execution is violated.

A memory model can be seen as a contract between the application and the underlying platform. Languages like C++11 and Java define a specific memory model. In this paper, the application is based on λ-calculus, which does not have the notion of memory, so there is no need to define the behavior of the memory at application-level. The RTS, however, does need a memory model. Since it is irrelevant for the application how the RTS is implemented, the choice of a memory model does not have to be a specific one, and could as well be a memory model that is easy to implement in hardware. It only has to be compatible with the requirements for executing a functional language. Next, the memory operations are derived from what happens during program execution.

4.2 A λ-Term's Life

Every term has the following sequence of phases during its lifetime:

(1) *Allocation on the worker's heap*
A term is either local or global, depending on the context in which it is created. They can share the same heap, but accessing a global term requires attention regarding memory consistency, which is discussed in a moment.

(2) *Initialization of the memory*
In our case, the constructor of the C++ object does this.

private access:

The term is only accessed by the owning thread. The term's content is constant after initialization.

(3) *Indirecting another term to this one*
A term is always a result of a reduction, so there exists a term that is replaced by the newly created one. Setting an indirection pointer to the new term will make it visible for other workers, which might follow the pointer.

(4) *Replace the term by the result of a reduction*
After β-reduction, the indirection pointer is set, which is the same operation as (3), but from a different perspective. A race condition exists, because multiple workers might reduce the same term simultaneously.

shared access:

The term is valid and globally accessible. Only the indirection field can be overwritten by concurrent threads.

(5) *Term dies*
When no pointer exists to this term, it becomes unused and can be garbage collected. Because the number of pointers to a term change at run-time, and it is subject to data races, this event is not detected during evaluation. Only during GC, the application graph is stable and can be analyzed.

(6) *Deallocation during GC*
At this point, the heap memory is freed.

private access:

The owning thread destructs and cleans up the term.

From this list, we can identify all operations that can be executed on a term, namely: *construction* (phase 1 and 2); *read* (during phase 3 and 4), where a worker reads the term after following a pointer to it; *indirect* (phase 4), where a worker sets the indirection pointer to the reduction result; and *destruction* (phase 6). For these operations, we discuss which guarantees, i.e. rules, are necessary to be implemented by the platform. Such a guarantee is something a worker thread can assume to be always valid.

4.3 Rules for Ordering and Practical Implementations

Although the intended behavior of the operations identified in the previous section is rather straightforward, the interaction between these rules is more complicated. In a similar manner as a memory model defines how reads and writes behave, the four operations on λ-terms have rules that define the required orderings to properly allow the execution of a functional program. This section defines these rules for all pairs of operations. Any platform should comply to these rules, by hardware support, software layers or a combination of both.

For clarification, we discuss a mapping of the operations and the rules to three different architectures: a PC, or x86 architecture, with corresponding strong memory model; a fictive architecture that implements the weak EC model; and a software cache coherent system. For software cache coherency, we used a multicore MicroBlaze setup [32]. In this system, every processor has its own non-coherent cache and is connected via an in-order interconnect to a single shared memory. The cache supports flushing and invalidating a specific cache line, where both operations remove the data from the cache, but the former writes dirty data back and the latter just discards it. Section 5 discusses the MicroBlaze system in more detail. For these architectures, different measures should be taken to implement the operations on λ-terms properly.

Construction of a term is obviously more than just a single read or write of memory. However, only the worker that creates the term can access this memory, because other workers do not have knowledge about its existence yet. So from a memory consistency point of view, this can be seen as a single operation. Any consecutive operation on the term should see the constructed term, which leads to the formulation of the following rule the memory subsystem must comply to:

RULE 1 (Construction). *Any worker that executes an operation on an existing term should observe that its construction has been completed.*

Although this sounds trivial, it means that the underlying system must make sure that the initialization of the term is completed and globally visible before a pointer to it is exposed to another worker. So, when another worker reads or sets the indirection pointer, the platform must make sure that the term's construction has been completed.

Generally, the term should be flushed and a fence should be executed after construction. In this context, a *flush* makes some effort to communicate changes to others as soon as possible—but not necessarily instantaneously. A *fence* (also known as a memory barrier) ensures that operations of the same processor before the fence should not be reordered with those after the fence. For x86, no special care has to be taken to guarantee this rule; a flush in hardware cache coherency is implicit and as stores are not reordered, the construction will be completed before a pointer to the new term is written[2]. For EC, the construction should be done with exclusive access to the memory, where the complete term is considered to be the shared object that is protected. In the case of software cache coherency, the term's memory should be flushed. Subsequent writes cannot be reordered with the preceding flush, effectively implementing a fence. Then, the background memory is up to date and all workers accessing the term will load that version in their cache.

[2] However, this is only valid for the memory model of the hardware. If an optimizing compiler is not aware of the importance of the order of these two writes, they might end up in different order. It is probably wise to declare such a term **`volatile`** or atomic.

When an indirection pointer of a term is set, the following rule must apply:

RULE 2 (Indirect). *Setting the indirection pointer from term t_1 to term t_2 is atomic and in globally total order with respect to other operations on term t_1 by the same worker and the construction of t_2.*

The restriction that writes should be atomic is usually already fulfilled by hardware, because pointers have (usually) the size of one machine word. If that is not the case, writing such a pointer will have overhead by locking and unlocking the related memory location. As described in Section 3.3, writing the indirection pointer twice does not harm the outcome of the program. Therefore, such writes do not have to be in total order, which is usually the case for memory models. The non-determinism by this data race is allowed, but should be solved by the GC later on.

For x86, an indirection pointer can just be written; all writes are in total order—although this is over-restrictive. Similarly, EC prescribes writing inside an exclusive access section, which not only guarantees a total order, but also prevents concurrent reads of the pointer. In a software cache coherency setup, the pointer can just be written. Although not essential for correctness, the cache line can be flushed afterwards to make the write visible for others.

Next, workers can read a term, possibly multiple times.

RULE 3 (Read). *Reads of a term are in a total order with respect to other operations on the same term by the same worker.*

In general, a read can just be executed. However, to receive updates of concurrent writes, a periodic flush is required. For x86, such a flush is done automatically by the hardware for every read. In EC, reads are done in shared (read-only) access, which always gets the value of the latest write in an exclusive block. Hence, reads always return the latest written value. However, it is not strictly necessary that writes are communicated that fast. Non-determinism by data races on a read and a concurrent write can be allowed, which in the worst case only result in some performance degradation by doing duplicate work. When it is more expensive to guarantee always having the latest data compared to incidentally doing work twice, an implementation for software cache coherency can just read the value from the cache and flush the cache periodically, during a GC run, for example. Then reads are mostly done from cache, but the cache can be somewhat out of date.

Because reads and indirections are only ordered per worker per term, interleavings of accesses of multiple terms or multiple workers are not defined. Moreover, workers might disagree on the observed orderings of these events. This is equal to Slow Consistency, which is easy to implement in hardware, as it does not enforce inter-processor orderings.

During GC, the program state is analyzed for dead terms. These terms should not be accessed afterwards.

RULE 4 (Garbage collection). *Before a worker destructs a term, all reads and indirections by any worker should be completed first.*

This also includes that after destruction, no worker should read or indirect the term anymore—otherwise the garbage analysis was faulty. Because the state of the memory is fixed during GC, any non-determinism in the indirection pointers can be solved by completing all outstanding writes first. This results in a single state of the application, which every worker agrees on. In general,

			new operation			
			c	i	r	d
previous operation	construction	c	×	1	1	1
	indirect	i	×	2	3	4
	read	r	×	2	3	4
	destruction	d	×	×	×	×

× Impossible

Table 1. Rules that pairs of operation on λ-terms are subject to

all outstanding writes should be flushed, after which a fence is required. For x86, the flush is implicit, but a fence is required by all participating workers before the terms are analyzed for liveliness. In an EC system, nothing special has to be done, because the exclusive sections already guarantee an up-to-date state. In the software cache coherency system, all terms have to be reconciled, so the dirty cache lines should be flushed.

Because memory is reused for another term, a copy of the old term of every worker should be discarded between destruction of the old and the construction of the new term. Since the destruction of a term is not communicated to other cores—only the worker owning the term decides that it is ready for GC, and keeping track of where old copies might be has some overhead—a practical implementation is to flush the complete cache during every global garbage collection. Different solutions might exist, depending on the method of garbage collection.

Finally, another data race is spotted: during garbage collection analysis. In a mark-sweep approach, multiple workers might write the flag of the same term concurrently. This data race can safely be ignored; all writers write the same value to the flag memory. Regardless of the ordering of these writes, the flag is set anyway, and usage of a mutex to protect it is not required. In the case of software cache coherency, a flush of the flags (or the whole cache for simplicity) after the marking phase is sufficient.

For every pair of executed operations, one of the four rules applies. Table 1 summarizes which rule applies for every pair of a previously executed operation and a new one.

As discussed above, the requirements on the hardware side are in general: support for a fence, which is only a processor-local ordering; a flush, which ensures visibility of the data for other processors, but no ordering is involved; and atomic (pointer) writes. Although a strict memory model satisfies these requirements, it might be beneficial to use a weaker model to allow the non-determinism as discussed above. Moreover, the same argument applies to hardware cache coherency; caches are always kept coherent, which incurs more work—and potentially even more problematic in future technologies, also more power—than strictly necessary. The next section presents experiments with systems with different memory architectures.

5. Experiments

We tested our functional language of Section 3.2 on two architectures. The first architecture is a hyperthreaded 12-core Intel Xeon system, which contains in this case 24 logical cores in total and runs Linux. On this system, the scalability of our atomic-free LambdaC++ and Haskell is tested.

The second architecture is a 32-core MicroBlaze system on FPGA. Every core of this homogeneous MicroBlaze system has a 16 KB instruction cache and 8 KB data cache. All cores share 128 MB DDR memory via an in-order interconnect [32]. Moreover, the MicroBlaze system does not have support for atomic RMW operations, but is capable to count micro-architectural events, including instructions, stall cycles, cache misses and memory operations. It runs a multi-threaded POSIX-like custom OS. The caches are not kept coherent in hardware, so the approach of Section 4 is applied to achieve software cache coherency.

The workload for the tests is delivered by applications from the parallel section of the Haskell NoFib Benchmark Suite [33]. We implemented five of them in LambdaC++, namely `coins`, `parfib`, `partak`, `prsa` and `queens`, and compare them to the Haskell versions in the experiments.

5.1 Scalability and Speedup

All Haskell applications are compiled with GHC 7.4.2 for the Intel platform. Our function language runs on both the Intel and the MicroBlaze platform. The first experiment measures the speedup of the application, depending on the number of cores used. Figure 3 shows the results for all applications and platforms. All applications have been run five times, and the measurements have been averaged.

In the figure, the speedup is shown, which is the multicore performance relative to the sequential run. So, with n cores, i.e. worker threads, a speedup of m means that the wall-clock execution time is m times less when n cores are utilized in parallel, compared to the execution time on one core. The execution of LambdaC++ requires about 400 instructions on average per created λ-term, including allocation, β-reduction, and garbage collection. Even though the absolute performance differs, the speedup shows similar behavior on x86. Both the Haskell and LambdaC++ versions show a close-to-linear speedup for about the first 10 cores[3]. After that, the execution time does not improve when using more cores.

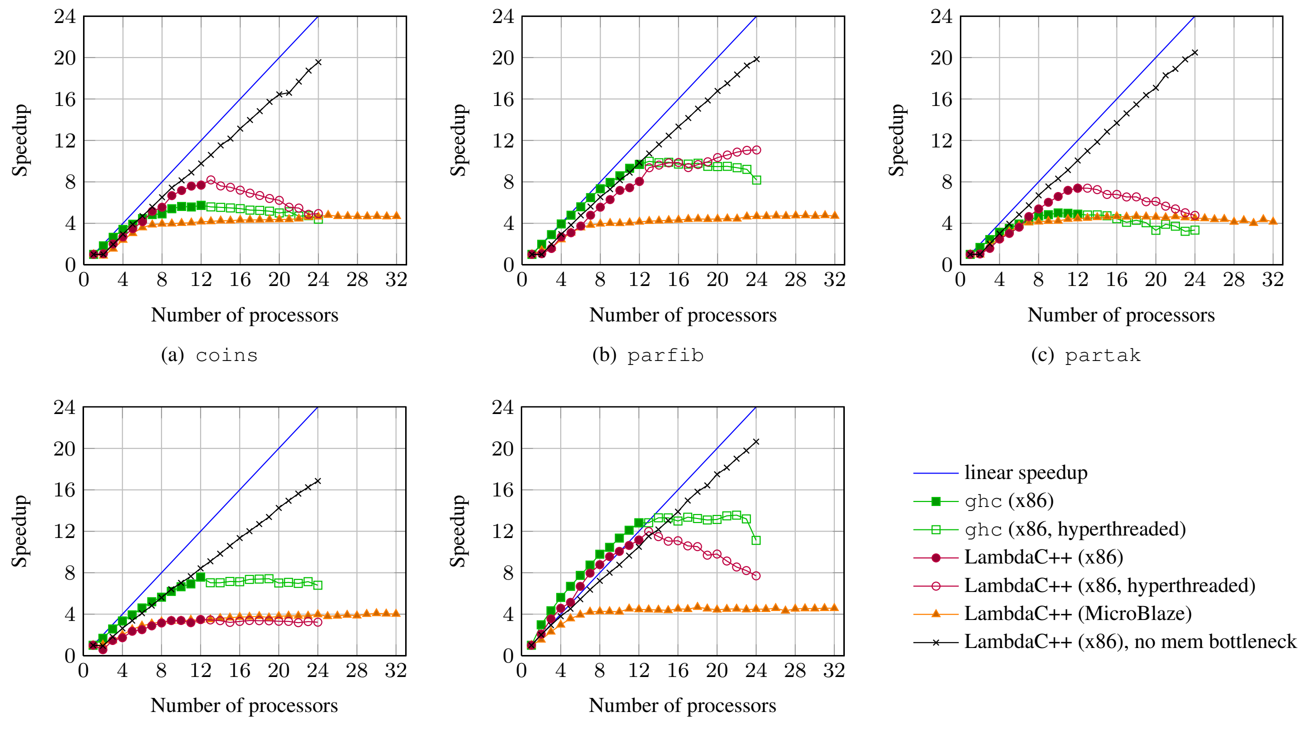
Linux's `perf` performance counters indicate that there is a memory bottleneck; the number of executed instructions is for every run the same—even the number of created λ-terms by LambdaC++ is independent of the number of workers—but the number of cycles the processors stall increased. The figure also shows the speedup when artificially compensated for this effect, which is labeled 'no mem bottleneck'. In that case, we calculated the speedup when the instructions, which are measured during the x86 runs of LambdaC++, would have the same number of stall cycles as during the sequential version. The straight line suggests that the speedup trend of the first ten cores is continued, at least up to 24 cores. This shows that the applications scale properly to many cores, although with some constant overhead. This also suggests that the non-determinism in these experiments does not result in performance loss by doubly calculated terms, although we cannot measure it precisely without influencing the execution.

The memory bottleneck is even more prominent on the MicroBlaze system. The bandwidth is saturated when eight cores are used. However, the same trend is visible; the workload scales properly to more cores and the same number of instructions is executed, but the processors just stall longer on every memory access. So, from a parallel workload point of view, our proposed approach of avoiding usage of locks and allowing data races seems to be viable.

5.2 Locality and Overhead

For every benchmark, we counted the amount of generated local function applications, local constant and all global terms. The ratio between local and global data for LambdaC++ running on the Intel platform is listed in Table 2. This table lists the measurements when using 12 cores, but the results are similar when another number of

[3] If looked very carefully, the reader might notice that having two cores for LambdaC++ does not improve the performance. This is due to the structure of the program. In the implementation, the programs build up a list. Then, all but one worker concurrently compute the contents of this list, and one worker is dedicated to post-processing the list in-order, e.g., to generate output. In practice, post-processing takes less time than computation, so with two workers, one worker computes, and the other waits for its result.



(a) `coins` (b) `parfib` (c) `partak` (d) `prsa` (e) `queens` Legend

Figure 3. Speedup of NoFib parallel benchmarks

benchmark	local applications[a]	local constants[a]	globals[a]
`coins`	0.418	0.582	$1.36 \cdot 10^{-4}$
`parfib`	0.379	0.621	$1.44 \cdot 10^{-4}$
`partak`	0.351	0.648	$5.47 \cdot 10^{-4}$
`prsa`	0.412	0.583	$4.97 \cdot 10^{-3}$
`queens`	0.445	0.555	$9.10 \cdot 10^{-5}$

[a] Fraction of sum of all global and local terms

Table 2. Generated terms during evaluation (LambdaC++, x86, 12 cores)

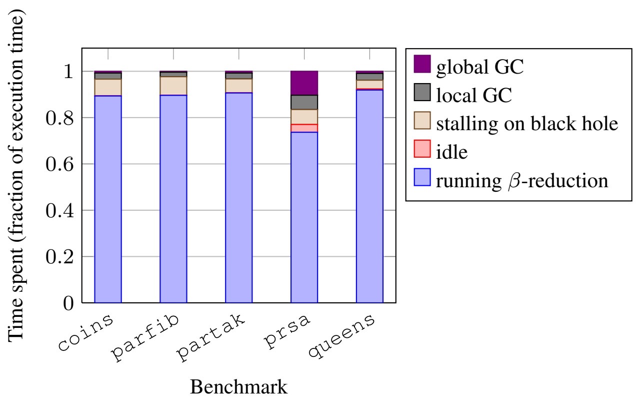


Figure 4. Time spent during execution (LambdaC++, x86, 12 cores)

cores is used. The table shows that the number of local terms is orders of magnitude higher than that of the global terms.

If all local terms can be kept local, traffic to main memory and the effects of the memory bottleneck will be reduced significantly. Although untested, a solution could involve having a (large) scratch-pad memory for every processor, and using this memory for all new local terms, i.e. the nursery of the GC. Anderson [24] reports that 99.8% of the data does not survive that private nursery stage, so they are dead at the successive GC. Such a modification to the RTS can be done transparently to the application. However, testing such a setup is left as future work.

Finally, the distribution of where time is spent during execution is measured. Figure 4 shows the most important states a worker can be in: global GC; local GC; stalling on a black hole, where another worker computes it; idle, because the work queue is empty; and running the application, which involves doing β-reductions. The time is the sum of of the time spent in such a phase, presented as a fraction of the combined total time of all workers. Only a small fraction is used for global GC, which is expected, because the number of global terms is much smaller than local onces. Interesting to see is that even local GC contributes only for 3.2% of the total execution time.

6. Conclusion

One of the hardware design issues of a multiprocessor platform is atomic global communication between cores, such as cache coherency and synchronization. In this paper, we showed that these hardware issues can be overcome at a different level. To this extend, we described a rather extreme example: a programming paradigm that allows an *atomic-free* implementation. Such an implementation does not rely on any read-modify-write operations or (mutex) locks, and does not rely on ordering guarantees of a strong memory model. We carefully introduce data races, even though the application keeps having a well-defined outcome.

For this, we implemented a functional language that strictly follows the properties of λ-calculus. Since the language is single-assignment, synchronization is simplified. Expressions that can be evaluated concurrently, can safely be pushed onto and popped from a work queue, without proper synchronization. When work is lost due to a race condition during the push, it will eventually be calculated when required. Moreover, because the evaluation of an expression in λ-calculus always gives the same result, multiple workers might evaluate expressions concurrently, and the doubly calculated results are just garbage collected.

Based on the programming paradigm, we derived ordering rules the memory subsystem must adhere to. These rules are weaker than implemented by x86's memory model, but can also be implemented as software cache coherency. Experiments on a 24-core Intel and a 32-core MicroBlaze architecture show that regardless of the number of worker threads and non-determinism in the execution, the amount of β-reductions and overhead appears to remain the same, which gives a good speedup when using more cores.

Atomic-free functional programs relax requirements on the hardware. This exemplifies the relation between programming abstraction and hardware platform, where the programming approach can reduce hardware complexity.

References

[1] G. Blake, R. Dreslinski, and T. Mudge, "A survey of multicore processors," *Signal Processing Magazine, IEEE*, vol. 26, pp. 26–37, 2009.

[2] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. Adve, V. Adve, N. Carter, and C.-T. Chou, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, 2011, pp. 155–166.

[3] F. Petrot, A. Greiner, and P. Gomez, "On cache coherency and memory consistency issues in NoC based shared memory multiprocessor SoC architectures," in *9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, 2006, pp. 53–60.

[4] J. Howard, S. Dighe, Y. Hoskote *et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, 2010, pp. 108–109.

[5] S. V. Adve and H.-J. Boehm, "Memory models: a case for rethinking parallel languages and hardware," *Commun. ACM*, vol. 53, pp. 90–101, 2010.

[6] E. Lee, "The problem with threads," *Computer*, vol. 39, 2006.

[7] Clojure, 2007. [Online]. Available: http://clojure.org

[8] C. Grelck, "Shared memory multiprocessor support for sac," in *Implementation of Functional Languages*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, vol. 1595, pp. 38–53.

[9] S. Marlow, S. Peyton Jones, and S. Singh, "Runtime support for multicore Haskell," *SIGPLAN Not.*, vol. 44, pp. 65–78, 2009.

[10] S. P. Jones, A. Gordon, and S. Finne, "Concurrent Haskell," in *Proceedings of the 23 rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, vol. 21, no. 24, 1996, pp. 295–308.

[11] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow, "Data parallel Haskell: a status report," in *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, ser. DAMP '07, 2007, pp. 10–18.

[12] House, 2006. [Online]. Available: http://programatica.cs.pdx.edu/House/

[13] J. Armstrong, R. Virding, C. Wikström, and M. Williams, "Concurrent programming in ERLANG," 1993.

[14] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí, "Parallel functional programming in Eden," *Journal of Functional Programming*, vol. 15, pp. 431–475, 2005.

[15] K. C. Sivaramakrishnan, L. Ziarek, R. Prasad, and S. Jagannathan, "Lightweight asynchrony using parasitic threads," in *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, ser. DAMP '10, 2010, pp. 63–72.

[16] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, 1998, ISBN 1558603433.

[17] A. Boeijink, P. K. F. Hölzenspies, and J. Kuper, "Introducing the PilGRIM: A processor for executing lazy functional languages," in *Implementation and Application of Functional Languages*, ser. Lecture Notes in Computer Science. Springer, 2011, vol. 6647, pp. 54–71.

[18] A. Bhattacharjee, G. Contreras, and M. Martonosi, "Parallelization libraries: Characterizing and reducing overheads," *ACM Trans. Archit. Code Optim.*, vol. 8, pp. 5:1–5:29, 2011. [Online]. Available: http://doi.acm.org/10.1145/1952998.1953003

[19] J. J. Tithi, D. Matani, G. Menghani, and R. A. Chowdhury, "Avoiding locks and atomic instructions in shared-memory parallel BFS using optimistic parallelization," in *Proceedings of the Workshop on Multithreaded Architectures and Applications (MTAAP 2013)*, 2013, pp. 1628–1637.

[20] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free irregular computations on GPUs," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ser. GPGPU-6, 2013, pp. 96–107.

[21] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, 2008.

[22] A. Church and J. B. Rosser, "Some properties of conversion," *Transactions of the American Mathematical Society*, vol. 39, no. 3, pp. 472–482, 1936.

[23] T. Johnsson, "Lambda lifting: Transforming programs to recursive equations," in *Functional Programming Languages and Computer Architecture*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1985, vol. 201, pp. 190–203.

[24] T. A. Anderson, "Optimizations in a private nursery-based garbage collector," in *Proceedings of the 2010 international symposium on Memory management*, ser. ISMM '10, 2010, pp. 21–30.

[25] R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall, Jan. 2012, iSBN 978-1-4200-8279-1.

[26] F. Henderson, "Accurate garbage collection in an uncooperative environment," in *Proceedings of the 3rd international symposium on Memory management*, ser. ISMM '02, 2002, pp. 150–156.

[27] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Commun. ACM*, vol. 17, pp. 453–455, 1974.

[28] J. H. Rutgers, M. J. G. Bekooij, and G. J. M. Smit, "Portable Memory Consistency for software managed distributed memory in many-core SoC," in *20th Reconfigurable Architectures Workshop (RAW 2013)*, 2013, pp. 212–221.

[29] M. Herlihy, "A methodology for implementing highly concurrent data objects," *ACM Trans. Program. Lang. Syst.*, vol. 15, pp. 745–770, 1993.

[30] Intel, "Intel 64 architecture memory ordering white paper," SKU 318147-001, 2007.

[31] B. Bershad, M. Zekauskas, and W. Sawdon, "The Midway distributed shared memory system," in *Compcon Spring '93, Digest of Papers.*, 1993, pp. 528–537.

[32] J. H. Rutgers, M. J. G. Bekooij, and G. J. M. Smit, "Evaluation of a connectionless NoC for a real-time distributed shared memory many-core system," in *Proceedings of the 15th Euromicro Conference on Digital System Design, DSD 2012*, 2012, pp. 727–730.

[33] NoFib Haskell Benchmark Suite, 2010. [Online]. Available: http://darcs.haskell.org/nofib/