University of Glasgow

School of Computing Science

Concurrency

Advanced Operating Systems (M)

Lecture 5

Colin Perkins | https://csperkins.org/ | Copyright © 2018 | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Lecture Outline

- Concurrency and the Implications of multicore systems
  - Memory models
  - Concurrency, threads, and locks – and their limitations
  - Alternative approaches – the multi-kernel model
- Managing concurrency using transactions
  - Programming model
  - Integration into programming languages

Colin Perkins | https://csperkins.org/ | Copyright © 2018

Concurrency and the Implications of Multicore Systems

Memory and Multicore Systems

- Hardware trends: multicore with non-uniform memory access
- Cache coherency is expensive, since the cores communicate by message passing and memory is remote

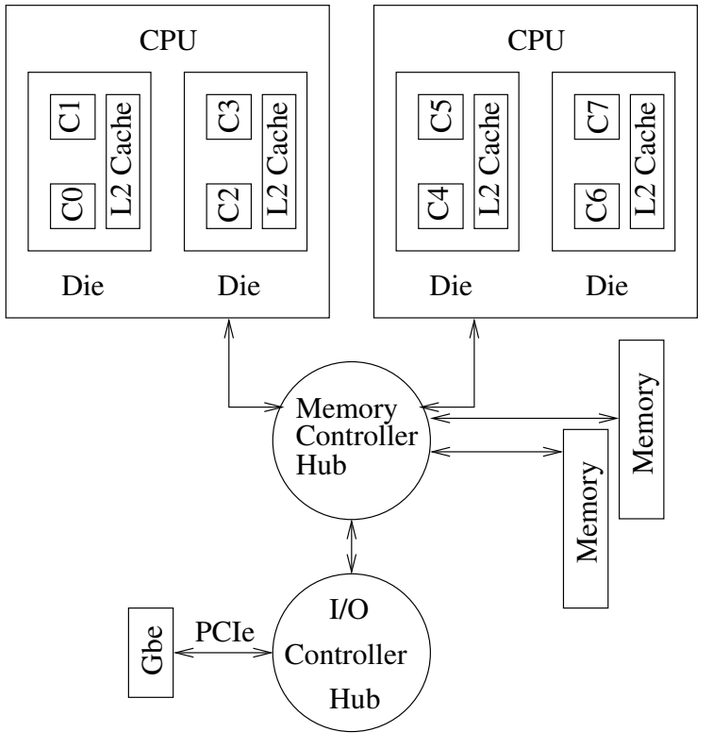


Figure 1. Structure of the Intel system

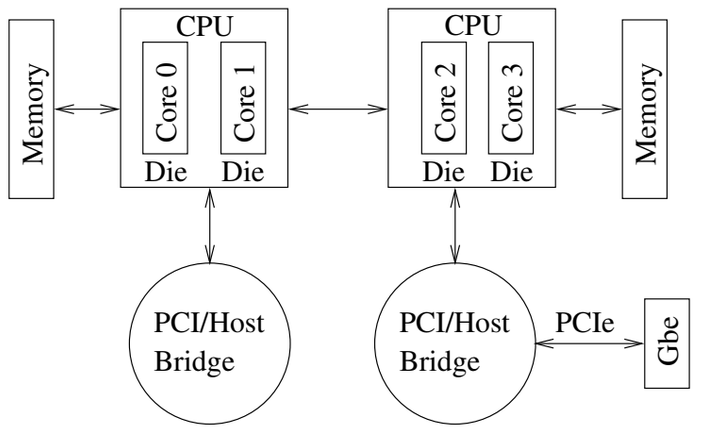


Figure 2. Structure of the AMD system

A. Schüpbach, *et al.*, Embracing diversity in the Barrelfish manycore operating system. Proc. Workshop on Managed Many-Core Systems, Boston, MA, USA, June 2008. ACM.

Colin Perkins | https://csperkins.org/ | Copyright © 2018

Multicore Memory Models

- When do writes made by one core become visible to other cores?
  - Prohibitively expensive for all threads on all core to have the exact same view of memory (“sequential consistency”)
  - For performance, allow cores inconsistent views of memory, except at synchronisation points; introduce synchronisation primitives with well-defined semantics
  - Varies between processor architectures – differences generally hidden by language runtime, *provided language has a clear memory model*

BY ND

Multicore Memory Models

- Memory Model defines space in which language runtime and processor architecture can innovate, without breaking programs
  - Synchronisation between threads occurs only at well-defined instants; memory may appear inconsistent between these times, if that helps the processor and/or runtime system performance
  - Without well-defined memory model, cannot reason about lock-based code
  - Essential for portable code using locks and shared memory

BY ND

Example: Java Memory Model

- Java has a formally defined memory model

[Somewhat simplified: see Java Language Specification, Chapter 17, for full details http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf]

- Between threads:
  - Changes to a field made by one thread are visible to other threads if:
    - The writing thread has released a synchronisation lock, and that same lock has subsequently been acquired by the reading thread (writes with lock held are atomic to other locked code)
    - If a thread writes to a field declared `volatile`, that write is done atomically, and immediately becomes visible to other threads
    - A newly created thread sees the state of the system as if it had just acquired a synchronisation lock that had just been released by the creating thread
    - When a thread terminates, its writes complete and become visible to other threads
  - Access to fields is atomic
    - i.e., you can never observe a half-way completed write, even if incorrectly synchronised
    - Except for `long` and `double` fields, for which writes are only atomic if field is `volatile`, or if a synchronisation lock is held
- Within a thread: actions are seen in program order

CC BY ND

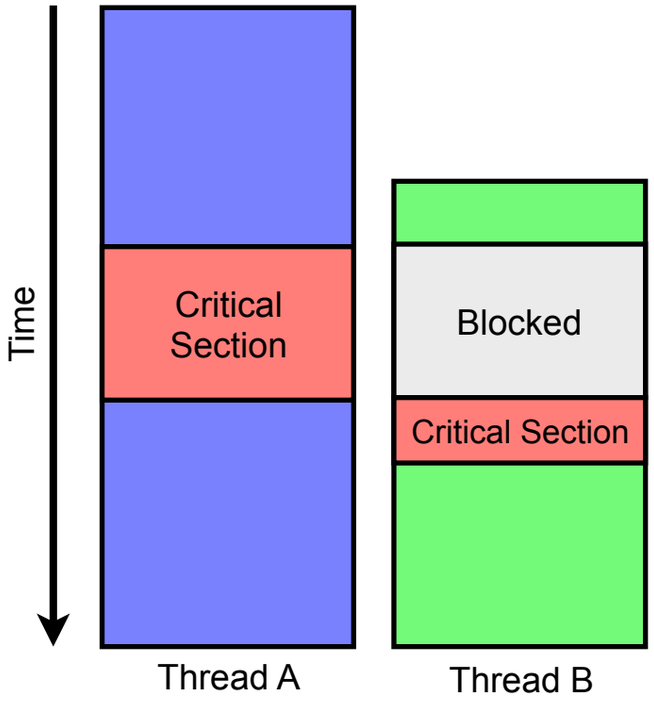
Multicore Memory Models

- Java is unusual in having such a clearly-specified memory model
  - Other languages are less well specified, running the risk that new processor designs can subtly break previously working programs
  - C and C++ have historically had *very* poorly specified memory models – latest versions of standards address this, but not widely used

Colin Perkins | https://csperkins.org/ | Copyright © 2018

Concurrency, Threads, and Locks

- Operating systems expose concurrency via *processes* and *threads*
  - Processes are isolated with separate memory areas
  - Threads share access to a common pool of memory
- The processor/language memory models specify how concurrent access to shared memory works
  - Generally enforce synchronisation via explicit locks around critical sections (e.g., Java synchronized methods and statements; pthread mutexes)
- Very limited guarantees about unlocked concurrent access to shared memory



Colin Perkins | https://csperkins.org/ | Copyright © 2018

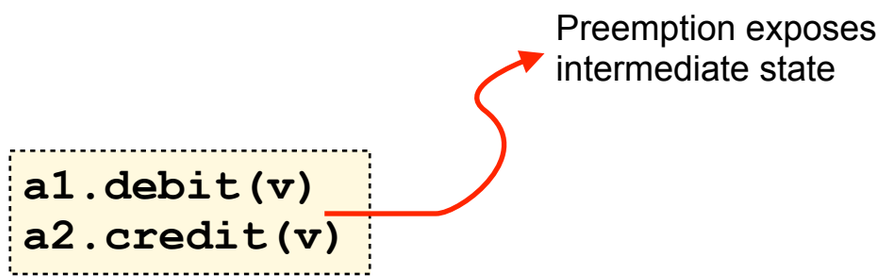
Limitations of Lock-based Concurrency

- Major problems with lock-based concurrency:
  - Difficult to define a memory model that enables good performance, while allowing programmers to reason about the code
  - Difficult to ensure correctness when composing code
    - Difficult to enforce correct locking
    - Difficult to guarantee freedom from deadlocks
    - Failures are silent – errors tend to manifest only under heavy load
    - Balancing performance and correctness difficult – easy to over- or under-lock systems

CC BY ND

Composition of Lock-based Code

- Correctness of small-scale code using locks can be ensured by careful coding (at least in theory)
- A more fundamental issue: lock-based code does not compose to larger scale
  - Assume a correctly locked bank account class, with methods to credit and debit money from an account
  - Want to take money from `a1` and move it to `a2`, without exposing an intermediate state where the money is in neither account
  - Can't be done without locking all other access to `a1` *and* `a2` while the transfer is in progress
  - The individual operations are correct, but the combined operation is not



- This is lack of abstraction a limitation of the lock-based concurrency model, and cannot be fixed by careful coding
- Locking requirements form part of the API of an object

CC BY ND

Alternative Concurrency Models

- Concurrency increasingly important
  - Multicore systems now ubiquitous
  - Asynchronous interactions between software and hardware devices
- Threads and synchronisation primitives problematic
- Are there alternatives that avoid these issues?
  - Message passing systems and actor-based languages
  - Transactional memory coupled with functional languages (e.g., Haskell) for automatic rollback and retry of transactions

CC BY ND

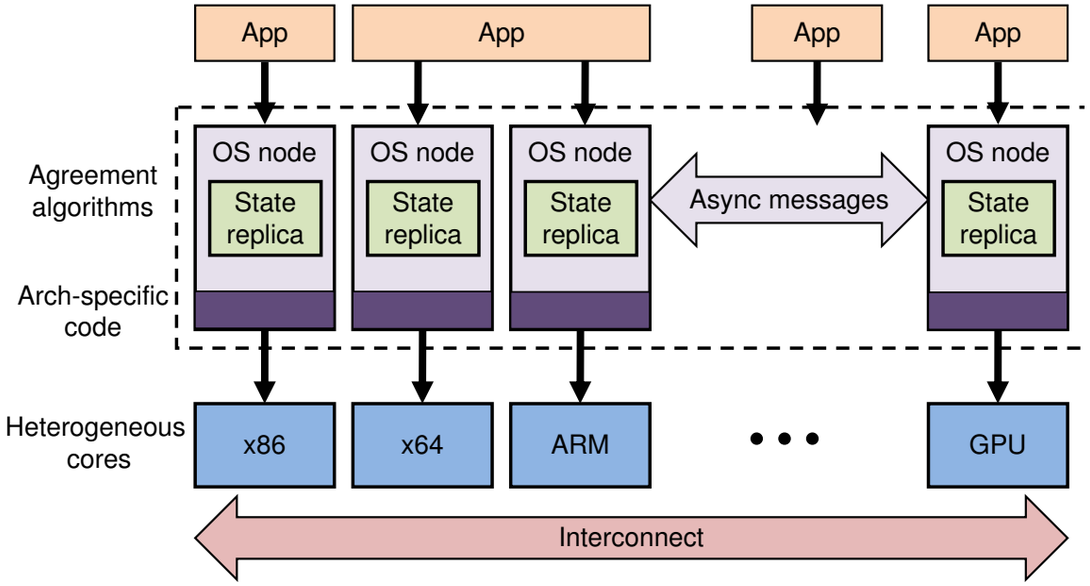
Implications for Operating System Design

- A single kernel instance may not be appropriate
  - Memory isn't shared – don't pretend it is!
  - There may be no single "central" processor to initialise the kernel
  - How to coordinate the kernel between peer processors?
- Multicore processors are increasing distributed systems at heart – can we embrace this?

BY ND

The Multi-kernel Model

- Build a distributed system that can use shared memory where possible as an optimisation, rather than a system that relies on shared memory
- The model is no longer that of a *single* operating system; rather a collection of cooperating kernels



Baumann *et al*, "The Multikernel: A new OS architecture for scalable multicore systems", Proc. ACM SOSP 2009. DOI 10.1145/1629575.1629579

- Three design principles for a multi-kernel operating system
  - Make all inter-core communication explicit
  - Make OS structure hardware neutral
  - View state as replicated instead of shared

CC BY ND

Principle 1: Explicit Communication

- Multi-kernel model relies on message passing
  - The only shared memory used by the kernels is that used to implement message passing (user-space programs can request shared memory in the usual way, if desired)
    - Strict isolation of kernel instances can be enforced by hardware
    - Share immutable data – message passing, not shared state
  - Latency of message passing is explicitly visible
    - Leads to asynchronous designs, since it becomes obvious where the system will block waiting for a synchronous reply
    - Differs from conventional kernels which are primarily synchronous, since latencies are invisible
  - Kernels become simpler to verify – explicit communication can be validated using formals methods developed for network protocols

CC BY ND

Principle 2: Hardware Neutral Kernels

- Write clean, portable, code wherever possible
  - Low-level hardware access is necessarily processor/system specific
  - Message passing is performance critical: should use of system-specific optimisations where necessary
  - Device drivers and much other kernel code can be generic and portable – better suited for heterogeneity
  - Highly-optimised code is difficult to port
    - Optimisations tend to tie it to the details of a particular platform
    - The more variety of hardware platforms a multi-kernel must operate on, the better it is to have acceptable performance everywhere, than high-performance on one platform, poor elsewhere
- Hardware is changing faster than system software

BY ND

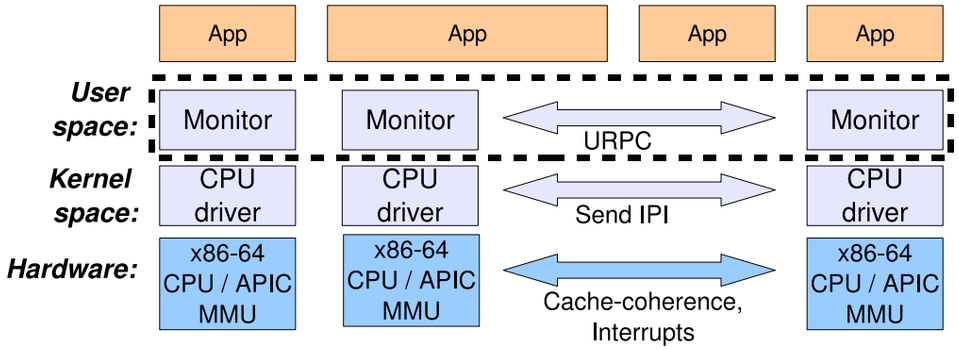
Principle 3: Replicated State

- A multi-kernel does not share state between cores
  - *All* data structures are local to each core
  - Anything needing global coordination must be managed using a distributed protocol
  - This includes things like the scheduler run-queues, network sockets, etc.
    - e.g., there is no way to list all running processes, without sending each core a message asking for its list, then combining the results
- A distributed system of cooperating kernels, not a single multiprocessor kernel

BY ND

Multi-kernel Example: Barrelfish

- Implementation of multi-kernel model for x86 NUMA systems
- CPU drivers
  - Enforces memory protection, authorisation, and the security model
  - Schedules user-space processes for its core
  - Mediates access to the core and associated hardware (MMU, APIC, etc.)
  - Provides inter-process communication for applications on the core
  - Implementation is completely event-driven, single-threaded, and non-preemptable
  - ~7500 lines of code (C + assembler)
- Monitors
  - Coordinate system-wide state across cores
- Applications written to a subset of the POSIX APIs



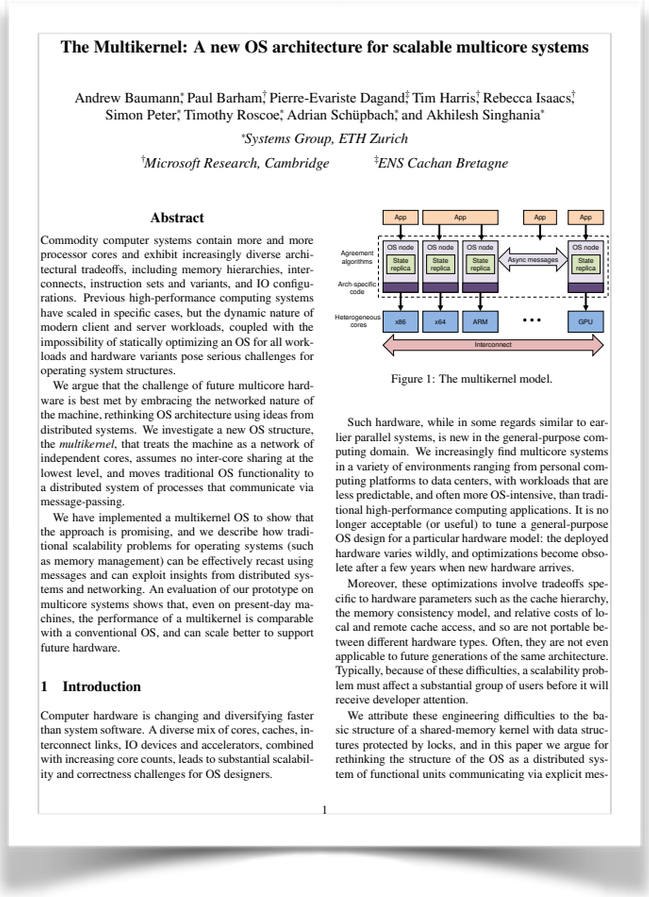
- Microkernel: network stack, memory allocation via capability system, etc., all run in user space
- Message passing tuned to details of AMD HyperTransport links and x86 cache-coherency protocols – highly system specific

CC BY ND

Further Reading for Tutorial

- A. Baumann *et al.*, “The Multikernel: A new OS architecture for scalable multicore systems”, Proc. ACM SOSP 2009. DOI:10.1145/1629575.1629579

- Barrelfish is clearly an extreme: a shared-nothing system implemented on a hardware platform that permits some efficient sharing
  - Is it better to start with a shared-nothing model, and implement sharing as an optimisation, or start with a shared-state system, and introduce message passing?
- Where is the boundary for a Barrelfish-like system?
  - Distinction between a distributed multi-kernel and a distributed system of networked computers?



The Multikernel: A new OS architecture for scalable multicore systems

Andrew Baumann*, Paul Barham†, Pierre-Evariste Dagand‡, Tim Harris†, Rebecca Isaacs†, Simon Peter*, Timothy Roscoe*, Adrian Schüpbach*, and Akhilesh Singhania*

*Systems Group, ETH Zurich

†Microsoft Research, Cambridge ‡ENS Cachan Bretagne

Abstract

Commodity computer systems contain more and more processor cores and exhibit increasingly diverse architectural tradeoffs, including memory hierarchies, interconnects, instruction sets and variants, and IO configurations. Previous high-performance computing systems have scaled in specific cases, but the dynamic nature of modern client and server workloads, coupled with the impossibility of statically optimizing an OS for all workloads and hardware variants pose serious challenges for operating system structures.

We argue that the challenge of future multicore hardware is best met by embracing the networked nature of the machine, rethinking OS architecture using ideas from distributed systems. We investigate a new OS structure, the *multikernel*, that treats the machine as a network of independent cores, assumes no inter-core sharing at the lowest level, and moves traditional OS functionality to a distributed system of processes that communicate via message-passing.

We have implemented a multikernel OS to show that the approach is promising, and we describe how traditional scalability problems for operating systems (such as memory management) can be effectively recast using messages and can exploit insights from distributed systems and networking. An evaluation of our prototype on multicore systems shows that, even on present-day machines, the performance of a multikernel is comparable with a conventional OS, and can scale better to support future hardware.

Figure 1: The multikernel model.

1 Introduction

Computer hardware is changing and diversifying faster than system software. A diverse mix of cores, caches, interconnect links, IO devices and accelerators, combined with increasing core counts, leads to substantial scalability and correctness challenges for OS designers.

Such hardware, while in some regards similar to earlier parallel systems, is new in the general-purpose computing domain. We increasingly find multicore systems in a variety of environments ranging from personal computing platforms to data centers, with workloads that are less predictable, and often more OS-intensive, than traditional high-performance computing applications. It is no longer acceptable (or useful) to tune a general-purpose OS design for a particular hardware model: the deployed hardware varies wildly, and optimizations become obsolete after a few years when new hardware arrives.

Moreover, these optimizations involve tradeoffs specific to hardware parameters such as the cache hierarchy, the memory consistency model, and relative costs of local and remote cache access, and so are not portable between different hardware types. Often, they are not even applicable to future generations of the same architecture. Typically, because of these difficulties, a scalability problem must affect a substantial group of users before it will receive developer attention.

We attribute these engineering difficulties to the basic structure of a shared-memory kernel with data structures protected by locks, and in this paper we argue for rethinking the structure of the OS as a distributed system of functional units communicating via explicit mes-

1

CC BY ND

Managing Concurrency Using Transactions

Managing Concurrency via Transactions

- Transactions for managing concurrency
- Programming model
- Integration into Haskell
- Integration into other languages
- Discussion

Colin Perkins | https://csperkins.org/ | Copyright © 2018

Transactions for Managing Concurrency

- An alternative to locking: use *atomic transactions* to manage concurrency
  - A program is a sequence of concurrent atomic actions
  - Atomic actions succeed or fail in their entirety, and intermediate states are not visible to other threads
  - The runtime must ensure actions have the usual ACID properties:
    - **A**tomicity – all changes to the data are performed, or none are
    - **C**onsistency – data is in a consistent state when a transaction starts, and when it ends
    - **I**solation – intermediate states of a transaction are invisible to other transactions
    - **D**urability – once committed, results of a transaction persist

```
atomic {
  a1.debit(v)
  a2.credit(v)
}
```

- Advantages:
  - Transactions can be composed arbitrarily, without affecting correctness
  - Avoid deadlock due to incorrect locking, since there are no locks

CC BY ND

Programming Model

- Simple programming model:
  - Blocks of code can be labelled `atomic {…}`
  - Run concurrently and atomically with respect to every other `atomic {…}` blocks – controls concurrency and ensures consistent data structures
- Implemented via optimistic synchronisation
  - A thread-local transaction log is maintained, records every memory read and write made by the atomic block
  - When an atomic block completes, the log is *validated* to check that it has seen a consistent view of memory
  - If validation succeeds, the transaction *commits* its changes to memory; if not, the transaction is rolled-back and retried from scratch
  - Progress may be slow if *conflicting* transactions cause repeated validation failures, but will eventually occur

Colin Perkins | https://csperkins.org/ | Copyright © 2018

Programming Model – Consequences

- Transactions may be re-run automatically, if their transaction log fails to validate
- Places restrictions on transaction behaviour:
  - Transactions must be referentially transparent – produce the same answer each time they're executed
  - Transactions must do nothing irrevocable:

```
atomic(n, k) {
  doSomeStuff()
  if (n > k) then launchMissiles();
  doMoreStuff();
}
```

    - Might launch the missiles multiple times, if it gets re-run due to validation failure caused by `doMoreStuff()`
    - Might accidentally launch the missiles if a concurrent thread modifies *n* or *k* while the transaction is running (this will cause a transaction failure, but too late to stop the launch)
- These restrictions must be enforced, else we trade hard-to-find locking bugs for hard-to-find transaction bugs

CC BY ND

Controlling I/O

- Unrestricted I/O breaks transaction isolation
  - Reading and writing files
  - Sending and receiving data over the networks
  - Taking mouse or keyboard input; changing the display
- Require language control of when I/O is performed
  - Remove global functions to perform I/O from the standard library
  - Add an *I/O context* object, local to `main()`, passed explicitly to functions that need to perform I/O
    - Compare sockets, that behave in this manner, with file I/O that typically does not
  - I/O functions (e.g., `printf()` and friends) then become methods on the I/O context object
  - The I/O context is not passed to transactions, so they cannot perform I/O
  - Example: the IO monad in Haskell

CC BY ND

Controlling Side Effects

- Code that has side effects must be controlled
  - Pure and referentially transparent functions can be executed normally
  - Functions that only perform memory actions can be executed normally, *provided* transaction log tracks the memory actions and validates them before the transaction commits – and can potentially roll them back
    - A *memory action* is an operation that manipulates data on the heap, that could be seen by other threads
    - Tracking memory actions can be done by language runtime (STM; software transactional memory), or via hardware enforced transactional memory behaviour and rollback
- Similar principle as controlling I/O
  - Disallow unrestricted heap access – only see data in transaction context
  - Pass transaction context explicitly to transactions; this has operations to perform transactional memory operations, and rollback if the transaction fails to commit
  - Very similar to the state monad in Haskell

BY ND

Monadic STM Implementation (1)

- Monads → well-defined way to control side-effects in functional languages
  - A monad `M a` describes an action (i.e., a function) that, when executed, produces a result of type `a` performed in the context `M`
    - Along with rules for chaining operations in that context
  - A common use is for controlling I/O operations:
    - The `putChar` function takes a character, operates on the IO context to add the character, and returns nothing
    - The `getChar` operates on the IO context to return a character
    - The main function has an IO context, that wraps and performs other actions

```
putChar :: Char -> IO ()
getChar :: IO Char
```

- The definition of the I/O monad type ensures that a function that is not passed the IO context cannot perform I/O operations
- One part of a software transactional memory implementation: ensure type of the `atomic {…}` function does not allow it to be passed an IO context, hence preventing I/O

CC BY ND

Monadic STM Implementation (2)

- How to track side-effecting memory actions?
  - Define an STM monad to wrap transactions
  - Based on the state monad; manages side-effects via a `TVar` type
    - The `newTVar` function takes a value of type a, returns a new `TVar` to hold the value, wrapped in an STM monad
    - `readTVar` takes a `TVar` and returns an STM monad; when performed this returns the value of that `TVar`; `writeTVar` function takes a `TVar` and a value, returns an STM monad that can validate the transaction and commit the value to the `TVar`

```
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

- The `atomic {…}` function operates in an STM context and returns an IO context that performs the operations needed to validate and commit the transaction
  - The `newTVar`, `readTVar`, and `writeTVar` functions need an STM action, and so can only run in the context of an atomic block that provides such an action
  - I/O is prohibited within the transaction, since operations in `atomic {…}` don't have access to the I/O context

```
atomic :: STM a -> IO a
```

BY ND

Integration into Haskell

- Transactional memory is a good fit with Haskell
  - Pure functions and monads ensure transaction semantics are preserved
  - I/O and side-effects contained in `STM` monad of an `atomic {…}` block
    - The `TVar` implementation is responsible for tracking side effects
    - The `atomic {…}` block validates, then commits the transaction (by returning an IO monad action to perform the transaction)
  - Untracked I/O or side-effects cannot be performed within an `atomic {…}` block, since there is no way to access the IO monad directly
    - There is no IO monad in scope within the transaction, so code requiring one will not compile
    - A `TVar` requires an `STM` monad, but these are only available in an `atomic {…}` block; can't update a `TVar` outside a transaction, so can't break atomicity guidelines – Haskell doesn't allow unrestricted heap access via pointers, so can't subvert

CC BY ND

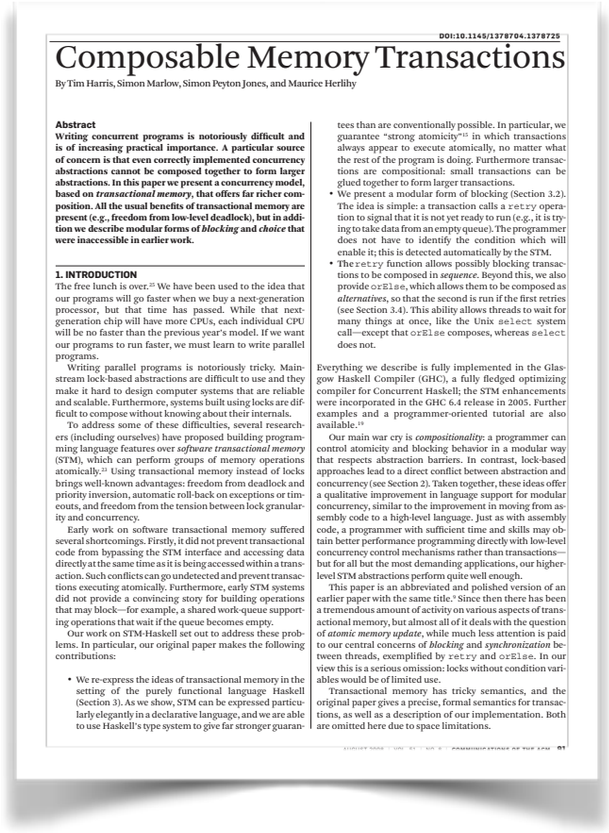
Integration into Other Languages

- STM in Haskell is very powerful – but relies on the type system to ensure safe composition and retry
- Integration into mainstream languages is difficult
  - Most languages cannot enforce use of pure functions
  - Most languages cannot limit the use of I/O and side effects
  - Transaction memory can be used without these, but requires programmer discipline to ensure correctness – and has silent failure modes
- Unclear if the transactional approach generalises to other languages

BY ND

Further Reading for Tutorial

- T. Harris, S. Marlow, S. Peyton Jones and M. Herlihy, “Composable Memory Transactions”, CACM, 51(8), August 2008. DOI:10.1145/1378704.1378725
  - http://www.cmi.ac.in/~madhavan/courses/pl2009/reading-material/harris-et-al-cacm-2008.pdf
- Is transactional memory a realistic technique?
  - Assumption: shared memory system, doesn't work with distributed and networked systems – is this true?
- Concurrent Haskell:
  - Monadic IO; do notation; IORefs; spawning threads
  - Type system separates state and stateless computation
- The STM interface
  - Composition; the `STM` monad, `atomic`, `retry`, and `orElse`, `TVars`
- Do its requirements for a purely functional language, with controlled I/O, restrict it to being a research toy?
- How much benefit can be gained from transactional memory in more traditional languages?



DOI:10.1145/1378704.1378725

Composable Memory Transactions

By Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy

Abstract

Writing concurrent programs is notoriously difficult and is of increasing practical importance. A particular source of concern is that even correctly implemented concurrency abstractions cannot be composed together to form larger abstractions. In this paper we present a concurrency model, based on *transactional memory*, that offers far richer composition. All the usual benefits of transactional memory are present (e.g., freedom from low-level deadlock), but in addition we describe modular forms of *blocking* and *choice* that were inaccessible in earlier work.

1. INTRODUCTION

The free lunch is over.[25] We have been used to the idea that our programs will go faster when we buy a next-generation processor, but that time has passed. While that next-generation chip will have more CPUs, each individual CPU will be no faster than the previous year's model. If we want our programs to run faster, we must learn to write parallel programs.

Writing parallel programs is notoriously tricky. Mainstream lock-based abstractions are difficult to use and they make it hard to design computer systems that are reliable and scalable. Furthermore, systems built using locks are difficult to compose without knowing about their internals.

To address some of these difficulties, several researchers (including ourselves) have proposed building programming language features over *software transactional memory* (STM), which can perform groups of memory operations atomically.[21] Using transactional memory instead of locks brings well-known advantages: freedom from deadlock and priority inversion, automatic roll-back on exceptions or timeouts, and freedom from the tension between lock granularity and concurrency.

Early work on software transactional memory suffered several shortcomings. Firstly, it did not prevent transactional code from bypassing the STM interface and accessing data directly at the same time as it is being accessed within a transaction. Such conflicts can go undetected and prevent transactions executing atomically. Furthermore, early STM systems did not provide a convincing story for building operations that may block—for example, a shared work-queue supporting operations that wait if the queue becomes empty.

Our work on STM-Haskell set out to address these problems. In particular, our original paper makes the following contributions:

- We re-express the ideas of transactional memory in the setting of the purely functional language Haskell (Section 3). As we show, STM can be expressed particularly elegantly in a declarative language, and we are able to use Haskell's type system to give far stronger guarantees than are conventionally possible. In particular, we guarantee “strong atomicity”[15] in which transactions always appear to execute atomically, no matter what the rest of the program is doing. Furthermore transactions are compositional: small transactions can be glued together to form larger transactions.
- We present a modular form of blocking (Section 3.2). The idea is simple: a transaction calls a `retry` operation to signal that it is not yet ready to run (e.g., it is trying to take data from an empty queue). The programmer does not have to identify the condition which will enable it; this is detected automatically by the STM.
- The `retry` function allows possibly blocking transactions to be composed in *sequence*. Beyond this, we also provide `orElse`, which allows them to be composed as *alternatives*, so that the second is run if the first retries (see Section 3.4). This ability allows threads to wait for many things at once, like the Unix `select` system call—except that `orElse` composes, whereas `select` does not.

Everything we describe is fully implemented in the Glasgow Haskell Compiler (GHC), a fully fledged optimizing compiler for Concurrent Haskell; the STM enhancements were incorporated in the GHC 6.4 release in 2005. Further examples and a programmer-oriented tutorial are also available.[19]

Our main war cry is *compositionality*: a programmer can control atomicity and blocking behavior in a modular way that respects abstraction barriers. In contrast, lock-based approaches lead to a direct conflict between abstraction and concurrency (see Section 2). Taken together, these ideas offer a qualitative improvement in language support for modular concurrency, similar to the improvement in moving from assembly code to a high-level language. Just as with assembly code, a programmer with sufficient time and skills may obtain better performance programming directly with low-level concurrency control mechanisms rather than transactions—but for all but the most demanding applications, our higher-level STM abstractions perform quite well enough.

This paper is an abbreviated and polished version of an earlier paper with the same title.[9] Since then there has been a tremendous amount of activity on various aspects of transactional memory, but almost all of it deals with the question of *atomic memory update*, while much less attention is paid to our central concerns of *blocking* and *synchronization* between threads, exemplified by `retry` and `orElse`. In our view this is a serious omission: locks without condition variables would be of limited use.

Transactional memory has tricky semantics, and the original paper gives a precise, formal semantics for transactions, as well as a description of our implementation. Both are omitted here due to space limitations.

CC BY ND