University of Glasgow

School of Computing Science

60 YEARS OF COMPUTING AT GLASGOW

Implications of Concurrency for Systems Programming

Advanced Operating Systems

Tutorial 5

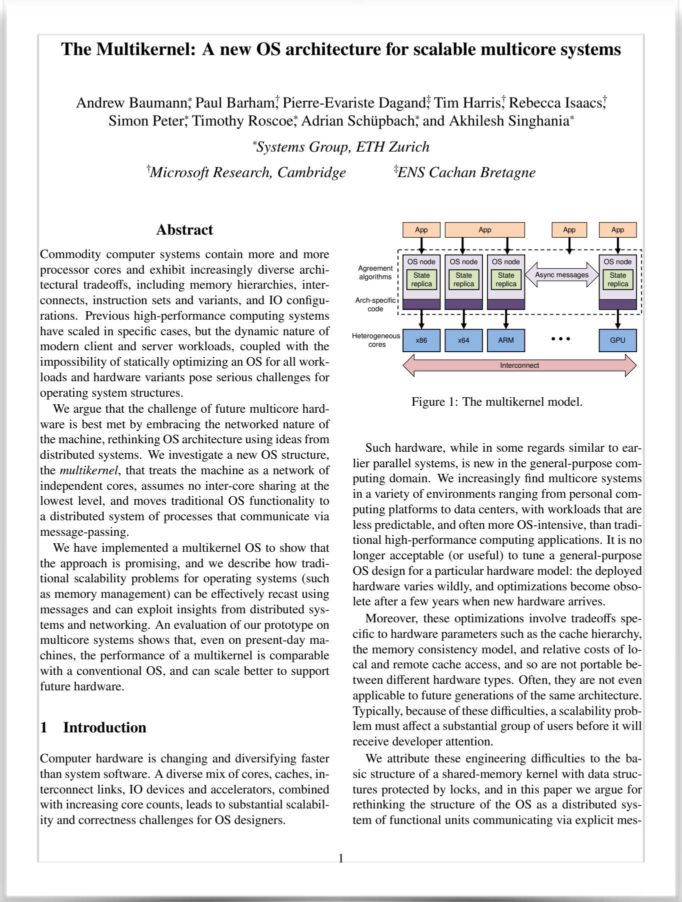
Review: Barrelfish

- Multicore memory models
  - When to memory writes become visible to other cores in a multicore system?
  - What are the synchronisation points?
  - The Java memory model
- Concurrency, threads, and locks
- Limitations of using lock-based concurrency – composition of lock-based code
- Alternative concurrency models
  - Message passing
  - Transactional memory
- Implications for operating system design
  - The multi-kernel model and Barrelfish

- Key points:
  - Shared-state concurrency using locks is not a good model
  - Alternative models exist, but change the way systems must be designed

Discussion: Barrelfish

- 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 an extreme: a shared-nothing system implemented on a hardware platform that permits some efficient sharing
  - Do you believe the arguments are hardware heterogeneity, ease and cost of messages vs. shared data?
  - Is explicit communication with replicated state a reasonable model?
  - Is performance reasonable?
  - 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?
- How does the design relate to Singularity?
- 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.

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.

Figure 1: The multikernel model.

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-

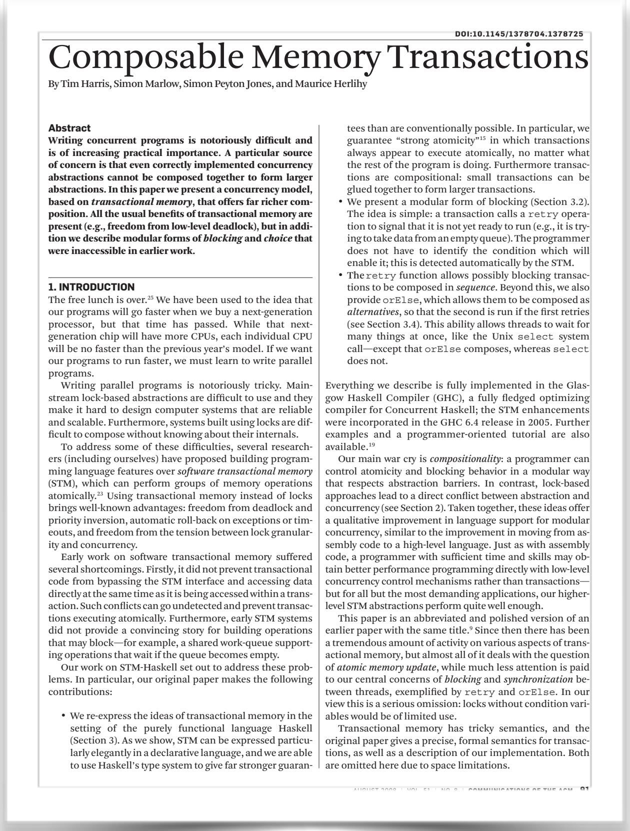
Review: Transactional Memory

- Concepts of transactions
  - ACID properties
  - Concurrent execution
  - Possible to compose transactions
- Implementation challenges
  - Controlling I/O operations
  - Controlling memory access – rollback and recovery
  - Implementation using monadic concepts
- Integration into Haskell
- Integration challenges for other languages

- Key points:
  - Understanding concepts of transactions
  - Understanding of implementation techniques in functional languages
  - Awareness of practical challenges

Discussion: Transactional Memory

- T. Harris, S. Marlow, S. Peyton Jones and M. Herlihy, “Composable Memory Transactions”, CACM, 51(8), August 2008. DOI:10.1145/1378704.1378725
- 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.[23] 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.

AUGUST 2008 | VOL. 51 | NO. 8 | COMMUNICATIONS OF THE ACM 81