Report from Dagstuhl Seminar 15021

Concurrent Computing in the Many-core Era

Edited by

Michael Philippsen[1], Pascal Felber[2], Michael L. Scott[3], and J. Eliot B. Moss[4]

1 **Universität Erlangen-Nürnberg, DE,** michael.philippsen@fau.de
2 **Université de Neuchâtel, CH,** pascal.felber@unine.ch
3 **University of Rochester, US,** scott@cs.rochester.edu
4 **University of Massachusetts – Amherst, US,** moss@cs.umass.edu

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 15021 "Concurrent computing in the many-core era". This seminar is a successor to Dagstuhl Seminars 08241 "Transactional memory: From implementation to application" and 12161 "Abstractions for scalable multicore computing", respectively held in June 2008 and in April 2012. The current seminar built on the previous seminars by notably (1) broadening the scope to concurrency beyond transactional memory and shared-memory multicores abstractions, (2) focusing on the new challenges and potential uses of emerging hardware support for synchronization extensions, and (3) considering the increasing complexity resulting from the explosion in heterogeneity.

Seminar January 5–9, 2015 – http://www.dagstuhl.de/15021
1998 ACM Subject Classification C.1.3 [Processor Architectures]: Other Architecture Styles – Heterogeneous (hybrid) systems, D.1.3 [Programming Techniques]: Concurrent Programming – Parallel Programming, D.3.3 [Programming Languages]: Language Constructs and Features – Concurrent programming structures, D.3.4 [Programming Languages]: Processors – Compilers, Memory Management, D.4.1 [Operating Systems]: Process Management – Synchronization, D.4.2 [Operating Systems]: Storage Management; H.2.4 [Database Management]: Systems – Transaction Processing
Keywords and phrases Multi-/many-core processors, Concurrent Programming, Synchronization, Transactional Memory, Programming Languages, Compilation
Digital Object Identifier 10.4230/DagRep.5.1.1

1 Executive Summary

Pascal Felber
Michael Philippsen
Michael L. Scott
J. Eliot B. Moss

Context and Motivations

Thirty years of improvement in the computational power of CMOS uniprocessors came to an end around 2004, with the near-simultaneous approach of several limits in device technology (feature scaling, frequency, heat dissipation, pin count). The industry has responded with ubiquitous multi-core processors, but scalable concurrency remains elusive

for many applications, and it now appears likely that the future will be not only massively parallel, but also massively heterogeneous.

Ten years into the multi-core era, much progress has been made. C and C++ are now explicitly parallel languages, with a rigorous memory model. Parallel programming libraries (OpenMP, TBB, Cilk++, CnC, GCD, TPL/PLINQ) have become mature enough for widespread commercial use. Graphics Processing Units support general-purpose data-parallel programming (in CUDA, OpenCL, and other languages) for a widening range of fields. Transactional memory appears likely to be incorporated into several programming languages. Software support is available in multiple compilers, and hardware support is being marketed by IBM and Intel, among others.

At the same time, core counts are currently lower than had once been predicted, in part because of a perceived lack of demand, and the prospects for increased core count over time appear to be constrained by the specter of dark silicon. Parallel programming remains difficult for most programmers, tool chains for concurrency remain immature and inconsistent, and pedagogical breakthroughs for the first- and second-year curriculum have yet to materialize. Perhaps most troublesome, it seems increasingly likely that future microprocessors will host scores or even hundreds of heterogeneous computational accelerators, both fixed and field-programmable. Programming for such complex chips is an exceptionally daunting prospect.

The goal of this Dagstuhl research seminar was to bring together leading international researchers from both academia and industry working on different aspects of concurrent computing (theory and practice, software and hardware, parallel programming languages, formal models, tools, etc.) in order to:

- assess the state of the art in concurrency, including formal models, languages, libraries, verification techniques, and tool chains;
- explore the many potential uses of emerging hardware support for transactional memory and synchronization extensions;
- envision next-generation hardware mechanisms;
- consider potential strategies to harness the anticipated explosion in heterogeneity; and
- investigate the interaction of synchronization and consistency with emerging support for low-latency byte-addressable persistent memory. (This last goal emerged late in the planning process, but became a major topic of discussion.)

Participants came from a wide variety of research communities, which seldom have the opportunity to meet together in one place. The seminar therefore provided a unique opportunity to focus diverse expertise on a common research agenda for concurrent computing on new generations of multi- and many-core systems.

Research Challenges

As part of this seminar, we specifically addressed the following challenges and open research questions, which are the focus of substantial investigation both in academia and in industry. These issues were addressed during the discussion at the workshop from the various perspectives of theory, concurrent algorithms, systems software, and microarchitecture.

The Future of Transactional Memory

With the introduction this past year of TM-capable commodity processors from IBM and Intel, TM research is increasingly turning to the question of how best to use the new hardware.

What can and cannot be accomplished with the simple interfaces currently available? What might be accomplished with the addition of non-transactional loads and/or stores within transactions? (And how should such stores behave?) What support might be needed for nested transactions or nested parallelism?

Given that machines without TM will exist for many years, and that HTM will remain bounded by constraints on capacity, associativity, etc., how should hardware and software transactions interact? What hardware extensions might facilitate the construction of hybrid systems? Can hardware transactions be used to accelerate STM? Is TM hardware useful for purposes other than TM?

Beyond these basic questions, how do we integrate TM into the concurrency tool chain? How does one debug a black-box atomic operation? How should TM be embedded into programming languages? Should speculation be visible to the programmer, or should it be hidden within the implementation? How large can transactions reasonably become? Should they remain primarily a means of building concurrent data structures, or should they expand to encompass larger operations-even system-level functions like I/O, thread/process interactions, and crash recovery? As implementations proliferate, are there reasonable models of correctness that move beyond opacity? How should we benchmark TM code? What performance counters should future TM hardware provide to profilers? What kind of infrastructure is needed to perform regression testing of transactional code?

Heterogeneity

GPUs are increasingly regarded as general-purpose computational resources, in platforms ranging from cell phones to supercomputers. Cell phones commonly include additional accelerators as well, for (de)compression, (de)encryption, and media transcoding. These and other accelerators (e.g., for linear algebra, pattern matching, XML parsing, or field-programmable functions) are likely to appear across the computing spectrum over the next few years.

In contrast to traditional (e.g., vector or floating-point) functional units, whose operations are uniformly short, and to traditional I/O devices, whose operations are uniformly long, accelerators can be expected to display a very wide range of response times. Long and variable response times suggest the need for resource management, to promote fair use across threads and applications. Short response times suggest the need for direct, user-level access-as already provided by GPU drivers from nVidia and (soon) AMD.

The prospect of contention for shared accelerators, accessed directly from user-level code, raises a host of questions for concurrent programming. How do we arbitrate shared access? Can traditional notions of locality be extended to accommodate heterogeneity? What happens to the tradeoff between local and remote computation when the alternatives use different instruction sets? What abstract models of progress/performance/time complexity are appropriate? Can operations that employ shared accelerators ever be considered non-blocking? How should we benchmark code that makes use of accelerators? What performance measures should heterogeneous architectures should provide to profilers? What kind of infrastructure is needed to perform regression testing in the face of heterogeneity?

Persistence

Exceptions like magnetic core and battery-backed RAM notwithstanding, mainstream computing has long maintained a firm separation between fast, volatile working memory and slow, non-volatile (persistent) storage. Emerging low-latency, byte-addressable technologies

like phase-change memory, memristors, and spin-torque-transfer memory bring this tradition into question. While near-term implementations may simply use low-latency nonvolatile memory as an accelerator for conventional file systems, alternative APIs may prove attractive. Specifically, it seems likely that future systems will give programmers the option of computing directly on persistent state, rather than reading it into working memory, using it there, and writing it out again. This possibility raises variants of many of the issues that have long concerned the concurrency community – consistency and atomicity in particular.

How should pointer-rich, non-file-based data be managed? Will we need automatic garbage collection? What will be the persistent analogues of nonblocking concurrent data structures? How will we ensure linearizability? Composability? A seemingly obvious option would add the 'D' (durability) to transactional memory's ACI (atomicity, consistency, and isolation). With little near-term prospect for integration of persistence and hardware TM, how will we minimize the overheads of persistent STM? What will the tradeoffs look like with respect to lock-based Lock-based programming models? What will be the division of labor between the operating system, runtime, and compiler? What will be the complexity models? Will we count "persistent accesses" the way we currently count remote memory accesses for concurrent objects in memory?

Pedagogy

Once upon a time, concurrency was a specialized topic in the undergraduate curriculum, generally deferred to the operating systems course, or to an upper-level elective of its own. Now it is an essential part of the training of every computer scientist. Yet there is surprisingly little consensus on where it belongs in the curriculum, and how it ought to be taught. Alternatives range from "concurrency first," to infusion throughout the curriculum, to more extensive coverage in a more limited number of courses.

While the principal focus of the seminar was on research issues, participants had the opportunity to share both intuition and experience in the teaching of concurrency, during a dedicated panel session and as part of informal discussions. The following questions were notably discussed. What works, for which kinds of students? What languages and tool chains should we use? What textbooks do we need? What role (if any) should be played by deterministic parallel languages and constructs? Are there approaches, particularly for introductory students, that can offer parallel speedup for important applications, without the full complexity of the general case? Can these approaches reasonably be "staged" into intro-level courses?

Organization of the Seminar

The seminar lasted 5 days, each composed of short scientific presentations, with ample time for discussions, and break-out sessions during which various open questions were discussed in sub-groups. The first day of the seminar started with a general introduction and forward-looking presentations on concurrency and the challenges raised by heterogeneity and virtualization.

Ten technical sessions, with short presentations from the participants, took place during the seminar on:

- locks and TM;
- C++ status and standards;
- memory models;
- memory management and persistence;

- performance tuning and verification;
- distributed concurrency and fault-tolerance;
- thoughts on concurrency and parallelism;
- HW and portability;
- compilers, runtimes, and libraries; and
- languages and systems.

They were complemented by break-out sessions on "dealing with heterogeneity", the "future of TM", and "persistence", as well as a plenary discussion on "virtualization". Finally, a panel discussion was organized on the topic of "teaching concurrency". The seminar concluded with an open discussion on the future of concurrency and the challenges that will need to be adddressed in coming years.

The topic of the sessions and their diversity illustrate the complexity of the challenges raised by concurrent computing on multi- and many-core systems. As one can expect from such prospective seminars, the discussions raised almost as many new questions as they provided answers on the addressed research challenges. Indeed, while there has been significant advances since the previous seminars (08241 and 12161), notably in terms of hardware support, few of the outstanding problems have been completely solved and new ones have emerged. For instance, hardware support for TM is now available in consumer CPUs but it cannot be used straightforwardly in real applications without relying on hybrid software/hardware strategies, notably to deal with the lack of progress guarantees and the possibility of spurious aborts.

As detailed in the rest of this report, the seminar has allowed the community to make significant progress on a number of important questions pertaining to concurrent computing, while at the same time defining a research agenda for the next few years. Participants provided very positive feedback following the seminar and expressed strong interest in follow-up events. Organizers strongly support the continuation of this series of seminars on concurrent computing, one of the most important and challenging fields in the era of multi- and many-core systems.

2 Table of Contents

3 Jump-Start Talks

3.1 Heterogeneous Concurrency

Michael L. Scott (University of Rochester, US)

It appears increasingly likely that future multi-/many-core processors will be highly heterogeneous, with cores that differ not only in average performance and energy consumption, but also in purpose, with instruction sets and micro-architecture specialized for such tasks as vector computation, compression, encryption, media transcoding, pattern matching, and XML parsing. We may even see ubiquitous FPGAs on-chip.

The time appears to be ripe for concurrency researchers to explore open questions in this area. How will we write programs for highly heterogeneous machines? Possible issues include:

- What will be the policies and mechanisms to allocate and manage resources (cycles, scratchpad memory, bandwidth)?
- Will we continue to insist on monolithic stacks, or will it make sense to allocate frames dynamically (sometimes, perhaps, in local scratchpad memory)?
- How will we dispatch work to other cores? Hardware queues? Flat combining?
- How will we wait for the completion of work on other cores? Spin? Yield? De-schedule? Perhaps we shouldn't wait at all, but rather ship continuations?
- When work can be done in more than one place, how will we choose among cores with non-trivial tradeoffs (in power, energy, time, or load)? How will we generate code for functions that may use different ISAs depending on where they run?
- What is the right division of labor between the programming language, the run-time system, the OS, and the hardware?
- What features would we like architects to build into future machines?

Notes *(collected by members of the audience)*

The purpose of this talk is to jump-start conversation.

- Background: Why don't we have 1000-core multi-cores? Because they would melt (at least if you used all the cores at the same time). As a result, we're looking at a future with billions of transistors on the chip, but many of them will have to be turned off ("dark silicon"). ([Charles E. Leiserson]: Or you could clock them down.) One way of dealing with dark silicon is to build special-purpose, customized circuits. Anything that is a non-trivial portion of execution time could have a specialized circuit. This is already happening in mobile, where we may also have cores with different computational/energy tradeoffs.
- Future programs may have to "hop" between cores. There's a progression of functionality: (1) FPU: pure, simple function (e.g., arctan); protection is not really an issue. (2) GPU: fire-and-forget rendering. (3) GPGPU: compute and return (with memory access); direct access from user space; one protection domain at a time. (4) First-class core: juggle multiple contexts safely (really not an accelerator anymore); preemption, multiprogramming.

- Challenges with these heterogeneous cores: (1) How do we arbitrate access to resources (cycles, scratchpad memory, bandwidth)? (2) How do we choose among cores – e.g., the faster core or the more efficient one? (3) How do we get access to systems services on "accelerators"? (4) How do we handle data movement? The "best" core may not be best if we have to move data between cores, or if the "best" core may be overloaded with other computation. (5) How do we manage heterogeneous ISAs?
- Challenges for concurrency: (1) How do I dispatch across cores? (HW queues? flat combining?) (1) GPGPU accesses are not mediated by the operating system! (2) How do we manage stacks? (contiguous stacks? linked frames instead?) (3) How should we envision accelerator-based computing? Call function and get result back? Or do we want to think of this as shipping continuations? (4) What language support do we need? It would be nice to avoid writing code in a different language for every accelerator. (5) How do we manage signaling across cores? Wake up threads, etc.?
- Unsupported hypotheses: The traditional kernel interface is not going to last. It cannot capture everything as a pthread anymore. We're already heading in this direction, with extensions for user-space threads, etc. We really need to rethink how an OS supports threads. Contiguous stacks may need to be replaced with chains of dynamically allocated frames. That will need compiler support. Accelerator cores are going to need first-class status, with direct access to OS services. A tree-structured dynamic call graph will be too restrictive. Rather than assume call and return, the accelerator may need to decide what to do with the result, and where to run the current context next.
- Discussion:
  [David F. Bacon]: We may be over-generalizing lessons from GPUs. Many accelerators may be "fixed-function." An FPU-like model may be easier for managing complexity. IBM has a coherent accelerator interface for access to FPGAs, etc. My view is that I don't necessarily want to use coherent memory on an FPGA. Please have cores use the same ISA.
  [Michael L. Scott]: There may be important differences between an "encryption accelerator" with a standard encryption algorithm and an engine that knows how to do XSLT that can apply arbitrary function at each node of a tree.
  [Stephan Diestelhorst]: Jumping off single ISA thought. ARM has Big/Little. Sometimes there is a functional block that you want to leave out of the "small" ISA – e.g., the vector unit. So you have mostly the same ISA, but a "wimpy" core may not implement big instructions.
  [David F. Bacon] & [Michael L. Scott]: You can use microprogramming to implement "beefy" instructions – e.g., serializing vector instructions. That's the standard way to provide ISA compatibility.
  [Stephan Diestelhorst]: Would you want the support to be OS-visible?
  [David F. Bacon]: I want as much compatibility as possible. Heterogeneity is giving you so much hassle already; anything you can do to minimize this is worth doing.
  [Charles E. Leiserson]: Much depends on what you're trying to do. There's already a lot of difference between multi-/many-core/desktop computing vs. embedded. Over time, there may be even more separation among these sorts of things. A cell phone does a lot more specialized computing than a laptop does. In the cloud, there is more of a push for things to be homogeneous – e.g., Amazon Web Services turns off clock-frequency changing.
  [Michael L. Scott]: On the other hand, AWS will provide you a CUDA engine if you ask for it. As a general principle, answers may be different in different contexts. I tend to see

more convergence, rather than divergence. Embedded and general-purpose computing may be getting more similar, rather than different.
[Charles E. Leiserson]: Maybe at some point we'll have cloud-specialized processors?
[Michael L. Scott]: I would guess that we'd get even more specialized options – a menu – of possible processor choices.
[Charles E. Leiserson]: But that's hard to manage. AWS offers different kinds of machines, but they're concerned with keeping the number of offerings small.
[J. Eliot B. Moss]: If you offer a uniform ISA, you've "solved" the compiler problem. This becomes more of a scheduling/processor binding problem. It makes it easier to migrate threads between processors – and thus no different at the level of writing a program.
[David F. Bacon]: Maybe I'm not saying that all processors have same ISA, but you do want to minimize ISA heterogeneity.
[Torvald Riegel]: ISA is one layer of complexity, but not the only one. Performance differences still call for different types of code. It doesn't matter if every core supports vector instructions: if an accelerator is slow at vector code, you may still use a completely different kind of code. ISA uniformity just pushes the problem up to a different layer of the stack.
[J. Eliot B. Moss]: Note that performance heterogeneity is the whole reason for heterogeneity in the first place, so ultimately that's something you can't hide.
[Sven-Bodo Scholz]: Having a homogeneous ISA does not really make compiler people's lives easier. If the compiler is the place where you choose whether you use a vector instruction or not, the compiler still needs to know how fast the instruction is. If the compiler knows about heterogeneity it may be easier to produce good code than it is with "simulated" homogeneity (of the ISA) that under the hood turns out not to be homogeneous.
[Michael L. Scott]: I'm curious about whether contiguous stacks are an albatross.
[Charles E. Leiserson]: Absolutely. There was an opportunity when we went to a 64-bit software stack where we could have made a change, but we didn't. Current calling conventions are even more optimized for linear stacks than previous generations.
[J. Eliot B. Moss]: There are existing, widely used systems that do linked stack-chunks. Not at an individual frame level, but we don't always have fully-linear stacks.
[Hans-J. Boehm]: GCC supports discontiguous stacks?
[Torvald Riegel]: Mostly, but it's not that fine grained. You just don't have to reserve everything up front. But I don't think that contiguous stacks are the problem here. We need to start at the language level. What if we have execution agents that are not full pthreads (with scheduler guarantees, etc.) Looking at this from the languages/libraries side of things may be more productive.
[Stephan Diestelhorst]: Go has discontiguous stacks, threadlets, etc. These concepts may exist at the language level, but we may still need to do something at lower levels to make them faster.
[Dave Dice]: We've tried split chunk stacks in the JVM, but it's hard to get it to work across multiple platforms. We tried to do it for 32-bit code because it's faster, but we ran out of stack space. The JIT may be able to optimize out checks to see if there's enough space in current stack chunks. The big problem is that JVM interacts with C code, so the thread model/execution model must somewhat mirror the pthreads world.
[David F. Bacon]: We saw the same thing in Jikes. It's easier in a single-language environment.
[Charles E. Leiserson]: Cilk did linked frames for everything 15 years ago. Overhead in

GCC was only 1–2%. But that stuff is better optimized today, so overheads might be much higher. At the same time, the flexibility it gives you is so great, it may be worth the tradeoff.
[J. Eliot B. Moss]: Maybe part of the performance issue is not the number of instructions but what happens in the cache.
[Charles E. Leiserson]: I don't think cache is a big issue. Stacks may have more of an effect on the TLB. Even if you're allocating stack frames off the heap, if you're using the memory in a stack-like manner, you still get pretty good cache locality. C and C++ work well with malloc, because when you free something, that's what you allocate next, while managed languages don't give you this benefit.
[David F. Bacon]: This is really an architectural artifact, because we can't say that stuff we're freeing doesn't need to be in the cache.

3.2 Concurrency in Virtual Machines

J. Eliot B. Moss (University of Massachusetts – Amherst, US)

Consider the problem faced by a designer of a virtual machine (instruction set and related facilities) intended to support a wide range of programming languages (static and dynamic, "low" level like C and "high" level like Haskell) on a range of hardware platforms. It is challenging enough to provide integer and floating point operations and basic control flow (compare, branch, call, return, exceptions). The situation is made rather more difficult with respect to concurrency. Not only is there variety around single-word atomic accesses and ordering of memory accesses, but "larger" abstractions such as messaging, block/wakeup, threads, and especially transactions, make it difficult to devise a suitable common denominator. We hope that discussion at this workshop will help advance our thinking about what a good collection of building blocks might be.

Notes *(collected by members of the audience)*

- Goal: define a language-independent and HW-independent intermediate representation (IR) that deals well with concurrency.
  "I have a very practical need to worry about this topic, because Tony Hosking and I are working on a new grant on building a new VM that deals well with concurrency. Problem setting: Language-level virtual machine that abstracts away hardware detail. Below the programming language – target representation for compilers. Would like to support a wide variety of languages. Similar to LLVM, but more targeted to managed languages. Similar to CLR. Want something close to JVM, but one that is less language-centric. Hard to target new languages to JVM, because you have to "bend over backwards" to fit things into JVM model. Support GC, threads, etc. Starting point: what should the instruction set/IR for the virtual machine look like? Points of agreement: arithmetic/logical instructions, primitive data types, call/return. But there's a lot we don't agree on." Assuming we do not want to impose specific high-level semantics such as race-freedom or a particular transaction model, what primitives or building blocks should we provide to the language implementer to allow them to roll their own and achieve good performance on various hardware? "Things nice to have for concurrency: some

single-word atomic primitives (CAS), guaranteed progress (FetchAndAdd) What about multi-word operations/transactions? No agreed-upon semantics in the languages, no standard support in hardware. So what primitives should a VM support? Goal: what are the key building blocks to build STM or exploit HTM." [Charles E. Leiserson]: This is not really VM specific: really common to any IR design problem.

- Some quick thoughts: (1) Support grouping operations together (some notion of "transactions"), (2) Should deal with ordering, (3) Should deal with policy (contention management), (4) Should handle multiple scales: single thread, hyperthreads, same socket, same box, more distributed.
- **Q**[Jose Nelson Ameral]: What do you have in mind when you say should deal with ordering? Is it TLS-style support, with single sequential order? **A**: Some notion of ordering between transactions. May want to specify a specific order in which transactions may commit. But may want to support general messaging, too. [Hans-J. Boehm]: If transactions are exposed at language level, also need to worry about memory visibility ordering between transactional code and non-transactional code. **Q**[Michael L. Scott]: Is it always the case that if B is ordered after transaction A, is it always the case that B will see every non-transactional operation that happens-before transaction A? **Q**[Torvald Riegel]: Are you adopting a data-race free requirement? Has implications for optimization. **A**: May be a good requirement. [Hans-J. Boehm]: But then you can't handle Java in its current form.
- Would also like to support semantics not just memory. Semantic conflicts, semantic undo/redo (open nesting, boosting, etc.). Not in hardware, but how can we integrate it into a system that perhaps uses HTM or other hardware support? How do we generate concurrency? What are the primitives? Fork/join? Do-across/Do-all? Communication/ordering? Wait/signal? Futures? If we put these things into a VM, could help support multiple languages. Though current project is not intended to support multiple languages simultaneously, due to library requirements. Scope: Tightly connected (cache coherent) to Loosely connected (distributed). How much can we assume is being handled in hardware vs. how much has to be managed by software. Likely to see less coherence in heterogeneous world. In summary, as a (language VM implementer): What should I offer to language implementers? As an abstraction of current/future hardware? To support current and future languages? [Michael L. Scott]: And what features should we suggest to architects?
- **Q**[Pascal Felber]: Do we want to have primitives for message passing in VM? **Q**[Maurice Herlihy]: What about existing languages? Common intersection between Ruby/ Python/Scala is difficult. **A**: Think more union, rather than intersection. [Michael L. Scott]: There is a lot to learn from CLR. Asked a lot of questions about how to support multiple languages. [Hans-J. Boehm]: Just don't copy the memory model! [Michael L. Scott]: Right. Learn from their mistakes, too. **A**: Worried that a model like CLR is mired in a past era of languages and hardware, rather than being more forward looking. [Michael L. Scott]: Useful mistake to learn from: wound up abandoning attempt to build managed language runtime on top of CLR. [Antony Hoskin]: My impression of CLR is that languages supported by CLR have a lot of commonality. I would worry that CLR constrains classes of languages that can be implemented. **A**: As an example of problems: what if VM allows arbitrary "pinning" of memory. That doesn't play well with some sorts of garbage collection. Would like to avoid that sort of hassle. [David F. Bacon]: Go allows interior pointers, which places a lot of limitations on what a VM can do. **A**: We support some limited use of interior pointers. But can't store them in arbitrary places,

send them around. Current VM supports something like structs, but anything higher level is supported more at the compiler/runtime level, instead of being baked in to IR. [Daniele Donetta]: Working on a similar project called Truffle. Attempt from Oracle to answer same questions. Support Java, JavaScript, Ruby, Python, R. But not focusing on interoperability. We answered first question (what to offer language implementers) offered API to language implementer. API is basically to write AST interpreter. But dealing with same problems of concurrency. **A**: We support "tagged" data type, and can implement optimizations where once the tag is tested, and you know what the type is, can generate specific code. [Pascal Felber]: Is there any idea of how much of a performance hit you would take by supporting multiple languages? The more general you are, the more compromises you have to make on performance/code generation. **A**: We would like to give "reasonable" performance, but not necessarily the best performance. The project is not trying to do the best JIT ever for a given IR. But the goal is for the IR to not substantially inhibit achieving good performance. Part of the project is to look at loop kernels and make sure that performance is within a few percent of GCC performance. Just because we support dynamic types doesn't mean that you have to use those dynamic types. [Antony Hoskin]: Idea is to regenerate new JIT-ed code as language-level compiler learns more about types. **A**: Different starting point than LLVM. Not looking at backend for heavily-optimized language. Instead, we're looking at situations where you're throwing new IR at VM (more-refined version of functions). [Charles E. Leiserson]: Right now, do-all is implemented as syntactic sugar on top of library code, so don't get the same optimizations as real for loops. How does that problem get solved in this context? What you want to do is optimize it as a real for loop, and then afterwards say "oh, this is parallel, so use language-specific parallelism construct to implement it." **A**: One way to phrase this question: how much of the optimization needs to happen before IR (language specific) and how much are we leaning on the language-independent JIT. [Charles E. Leiserson]: What you would want is some sort of callback: JIT implements strength-reduction/ code motion, then calls back to compiler for actual parallel language construct. **A**: Could imagine some sort of step-wise refinement. I see that it's only an integer, so I'll generate code with a test at the top, and if it turns out I get a non-integer, I'll call back to the language-level compiler. **Q**[Michael L. Scott]: One of the basic questions you have for a parallel for loop is whether the iterations are themselves schedulable tasks? Or are they units of work that you pass off to schedulable things? [Antony Hoskin]: We have some primitives for constructing schedulable things. We view this as a language-level issue.

3.3 Concurrency and Transactional Memory in C++: 50000 foot view

Hans-J. Boehm (Google – Palo Alto, US)

The 2011 C++ standard first added explicit thread support and a corresponding memory model to the language. This was refined in relatively minor ways in the 2014 version of the standard. This represents significant progress, but some difficult problems, mostly related to the definition of weak memory orders, remain.

Recent work of the committee has focused on the development of more experimental technical specifications. Specifications nearing completion include one for transactional

memory and one specifying a parallel algorithms library. The design of transactional memory constructs was described.

Notes *(collected by members of the audience)*

- Concurrency Study Group (ISO JTC1/SC22WG21/SG1); transactional memory is separate (SG5).
  Tend to be inventive; goal is technical specification capturing community consensus. It describes C++ language semantics, not implementation rules or allowable optimizations. It is not a formal mathematical specification or textbook.
- C++11: added threads API (benefits from lambda expressions), an atomic operations library that relaxes SC through specifying weaker models, and specifies memory model, that is shared variable semantics. Three important aspects are (1) sequential consistency for data-race-free programs, (2) undefined semantics otherwise, and (3) trylock() and wait() may spuriously fail/return – wait() (aside from lock release and re-acquisition) and failed trylock() do not have synchronization behavior. **Q**: How fast is the standard adapted by compilers? **A**: Rather fast, major compilers already comply to it before it gets standardized. **Q**: Is atomic limited in any way? **A**: Standardized optimizations for different base types, otherwise should be implementable with a lock.
- C++14: added rwlock, shared_timed_mutex, and some hand waving for known issues.
- There are a number of conspicuous holes: (a) memory_order_relaxed probably implemented correctly, but needs proper spec (out-of-thin-air is the problem with circular dependencies, but it is unclear what a dependency is) (b) memory_order_consume needs work, (c) async() beginner thread creation facility has a serious design flaw, (d) no concurrent data structures, and (e) incomplete synchronization library.
- Two optional additions to the standard may become candidates for inclusion: parallel/vector algorithms (STL plus a bit) and miscellaneous concurrency extensions: extendedfutures, latches, (OpenMP-style) barriers, atomic smart pointers.
- Longer-term: fix async(), fork-join parallelism, asynchronous computation without explicit continuations ("resumable functions"), low level waiting API: synchronic<T> to wait for a specific value change of a specific variable, more general vector parallelism, and various concurrent data structures.
- Further out: fix memory order spec, mix atomic and non-atomic operations on the same location, better specification of execution agents (beyond bare OS threads) and progress properties.
- Transactional Memory: Tech spec out for balloting. It is experimental: where SG could not decide, both options are included. In C++11 locks require lock ordering, but that's intractable with call backs and hard to define in heavily templatized environments. Use TM as a syntactical way to drop lock-order issues, not a performance point. Syntax: special basic block types synchronized . . . atomic_noexcept/cancel/commit . . . . Not a full replacement for mutexes. Interaction with condition variables is open question. atomic_* act the same in absence of non-transactional accesses or exceptions. TM-semantics: behave as if a global lock was held (but performance expected to be better).
- Different flavors: synchronized allows non-txn synchronization nested (including IO). (This is useful where a lock needs to be acquired inside the transaction in a rare case.) atomic_* has no support for non-tx synchronization. Shared semantics: no nesting, no exceptions; thus single-global lock semantics. Strongly atomic in the absence of data races. Issues around what synchronization is allowed within a transaction. atomic_* have different behavior when an exception is thrown inside the tx. atomic_commit

commits if exception is thrown; atomic_cancel unrolls on a throw (but hard to get the state right): exception is propagated from inside the transaction, but state is rolled back; needs closed nesting due to aborted inner child. atomic_noexcept disallows exceptions. Aborts are problematic. Synchronized vs. atomic_commit: same if code is compatible, but atomic_commit has the compiler check the synchronization freedom of the body (compiler can make stronger static guarantees). tx-safety is part of type. Functions can be declared transaction_safe to be included in atomic blocks.

- Remaining concerns: Optimization/synchronization removal: prove that single thread-local modification can drop the transaction (empty transactions have stronger semantics than no-op, idea to "lock" accessed objects rather); Should transactions logically lock individual objects rather than a single global lock? (Under the single global lock model, an empty transaction still has a semantic effect); Interesting cases: statics, memory allocation is legal inside of transactions in spite of synchronization, some dynamic checking remains for virtual functions.
- Specification keeps growing; more work needed in library interaction. There will be changes. Comments welcome on the draft specification: https://groups.google.com/a/isocpp.org/forum/#!forum/tm

4 Overview of Talks, sorted alphabetically by Speaker

4.1 Tuning X Choice of Serialization Policies

Jose Nelson Amaral (University of Alberta, CA)

Best-Effort Hardware Transactional Memory (BE-HTM) systems require non-speculative fallback policies, or serialization managers, in order to provide a guarantee of forward progress to each transaction. There are several choices of serialization managers that can be used to build a BE-HTM system and most serialization managers have one or more parameters that change their behavior. Several published studies compare two or more alternative serialization managers, but do not explore the tuning of these manager parameters. In this talk I will present evidence, based on experimentation with the IBM Blue Gene/Q machine, to support the claim that the tuning of parameters for a serialization manager is very important and that (1) a fair comparison of serialization managers must explore their tuning; and (2) tuning is essential for each new HTM design point and for each type of application target.

Notes *(collected by members of the audience)*

- A number of papers compare TM policies
- Information about Blue Gene packaging and HTM policy
  16 cores on a chip; L1 16 Kb; L2 32 Mb (where the magic happens!); most of the rest is not TM aware.
  Two modes: short-run and long-run modes; L2 must be aware of all accesses in a txn, so have writes bypass L1; long-run mode invalidates/flushes L1 completely before txn starts; associate a speculative ID with a running txn (there are a limited number of these: 128, may be more than hardware threads). Note: txns can survive OS calls, so this can be an issue in some applications; BG/Q supports orders of magnitude more speculative write state than other implementations.

Failure modes (transaction conflict, capacity overflow, attempt to perform an irrevocable action, design space); conflict detection granularity vs. storage available for speculative state. (Blue Gene allows unusually large transactions at finer granularity than many systems).

- Contention managers x Serialization managers

  If HTM fails, it doesn't tell you who you conflicted with, hence at some point need to use serial execution.

  Simplest policy: go serial if the number of retries exceeds a threshold. Some apps are not sensitive to the threshold, especially in short mode; some performance better (with sharp break points) with larger thresholds, some degrade.

  MaxRetry Policy: Serialize once a certain number of retries is exceeded.

  LimitMeanST policy: favor thread that does the most work, based on karma. Hard to do directly in HW, so track time spent in txn, and serialize if you exceed your max time budget. This often gives a sweet spot for long-run mode, but some apps still degrade, and short-run mode is perhaps more variable. Does any serialization manager dominate the others? No, depends on tuning parameter.

4.2 Complexity Implications of Memory Ordering

Hagit Attiya (Technion – Haifa, IL)

Joint work of Attiya, Hagit; Guerraoui, Rachid; Hendler, Danny; Kuznetsov, Petr; Levy, Smadar; Michael, Maged; Vechev, Martin; Woelfel, Philipp
Main reference H. Attiya, D. Hendler, P. Woelfel, "Trading Fences with RMRs and Separating Memory Models," submitted.

Compiler optimizations that execute memory accesses out of (program) order often lead to incorrect execution of concurrent programs. These re-orderings are prohibited by inserting costly fence (memory barrier) instructions. The inherent Fence Complexity is a good estimate of an algorithm's time complexity, as is its RMR complexity: the number of Remote Memory References the algorithm must issue.

Ensuring the correctness of objects supporting strongly non-commutative operations (e.g., stacks, sets, queues, and locks) requires to insert at least one read-after-write (RAW) fence. When write instructions are executed in order, as in the Total Store Order (TSO) model, it is possible to implement a lock (and other objects) using only one RAW fence and an optimal O(n log n) RMR complexity. However, when store instructions may be re-ordered, as in the Partial Store Order (PSO) model, there is an inherent tradeoff between fence and RMR complexities.

In addition to the main reference above, this talk is also based on [1, 2].

References

1 Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proc. of the 38th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (POPL'11), pp. 487–498, ACM, 2011. DOI: 10.1145/1926385.1926442.

2 Hagit Attiya, Danny Hendler, and Smadar Levy. An o(1)-barriers optimal RMRs mutual exclusion algorithm: Extended abstract. In *Proc. of the 2013 ACM Symp. on Principles of Distributed Computing* (PODC'13), pp. 220–229, ACM, 2013. ACM. DOI: 10.1145/2484239.2484255.

Notes *(collected by members of the audience)*

- Talk models processor influence on shared memory with a reordering buffer located between each processor and the memory.
  Out-of-order execution avoided with fences & atomic operations. Memory model gives abstract conditions on how reordering can happen. Many models and sets of models have been proposed – e.g., Sun's sequential consistency (SC), TSO, PSO, RMO hierarchy. Customary to think of SC as "the good one."
- First result: a mutex algorithm must include a R-W fence (= flush of the reordering buffer) or equivalent atomic operation [1]. Holds for various other non-commutative ops (queues, counters, ...).
- Not all memory accesses are equal – for example, the Bakery Algorithm needs O(1) fences but O(n) accesses, which unfortunately must be remote, that is, served from the shared memory, not a local cache, i.e., with global communication.
- Tournament tree gives O(log n) fences and remote references.
- Without store reordering, one can get by with O(log n) RMRs and O(1) fences [2]. Uses a tree to combine processes into a queue for the lock.
- With store reordering (e.g., PSO), one cannot optimize both RMRs and fences. We can illustrate this with a tree of varying fan-out. The number of levels f determines the number of fences. The fan-out times the number of levels determines the number of RMRs.
- One can prove this is optimal: when stores can be reordered, any mutex algorithm has an execution E (one in which every process gets through the CS once) in which $F_E log(R_E/F_E) \in \Omega(nlogn)$ The proof uses an encoding argument, which captures the order in which processes enter the critical section [Attiya, Hendler, and Woelfel, submitted].
- A nice corollary of this work: there is a complexity separation between TSO and PSO. This suggests that TSO is "nice" in a strong way – analogous to how we have traditionally thought of SC as "nice".
- Also F log (R/F) is $\Theta(nlogn)$, where F is the number fences and R is the number of remote references; this cost is for all n processes to acquire the mutex once.
- Lower bound was instructive in finding the algorithm that meets it.
- **Q**[Nir Shavit]: How much of this translates to search trees? **A**: Not yet clear. We could use a sharper definition of the objects to which the theorem applies.

4.3 Heterogeneous Computing: A View from the Trenches

David F. Bacon (Google – New York, US)

Based on experiences with the Liquid Metal project at IBM Research, I describe the challenges ahead for heterogeneous computing. While compiler and run-time technologies can significantly reduce the complexities, radically different hardware organizations will still require fundamentally different algorithms. This will limit improvements in programmer productivity and keep costs of heterogeneous systems significantly higher. Nevertheless, over the long term heterogeneity will inevitably pervade computing systems.

Notes *(collected by members of the audience)*

- Various kinds of heterogeneity (ISA, scale,performance (thin vs. fat cores), fundamental organization (e.g., CPU vs. GPU), implementation technology, interconnect, language/library, algorithm)
- What users want: single language, compiler deals with platforms transparently, run-time handles variations, code migrates between platforms and responds (dynamically?) to load / grain / input variations.
- IBM Liquid Metal project
  - Single language Lime: Java-like, integration of a degree of static-ness in a dynamic language, data-parallel operators, stream graphs of isolated tasks, fine-grained primitive types, compile-time evaluation, exclusion on a per-platform basis (certain features not implemented on certain platforms), common run-time.
  - Transparent loading and data movement
  - Dynamic replacement
- Reality check
  - Organization tends to dictate the algorithm. So multiple implementations needed even in a single language.
  - Scientific comparison impractical
  - More heterogeneity tends to lead to lower utilization
  - HW specialization subject to obsolescence
  - More specialization = more total cost of operation = lower value
- Example of a challenging situation: Arrays in registers (via scalar replacement) versus an indexed block RAM
- Whither heterogeneity: Dark silicon is our friend; adoption will be slow, due to external factors; algorithmic heterogeneity is inescapable; pressure to minimize variants will remain; things look good with a 30-year horizon.
- Overarching experience: heterogeneity is *really* hard to manage; worth our while to avoid it wherever possible, and shield application-level programmers from it wherever possible.

4.4 Scalable consistency in distributed systems

Annette Bieniusa (TU Kaiserslautern, DE)

Joint work of Bieniusa, Annette; Shapiro, Marc; Preguica, Nuno; Zawirski, Marek; Baquero, Carlos
URL https://syncfree.lip6.fr

Replicating dynamically updated data is a principal mechanism in large-scale distributed systems, but it suffers from a fundamental tension between scalability and data consistency. Eventual consistency sidesteps the synchronization bottleneck, but remains ad-hoc, error-prone, and difficult to prove correct.

In this talk, I introduced a promising approach to synchronization-free sharing of mutable data: consistent replicated data types (CRDTs). Complying to simple mathematical properties (namely commutativity of concurrent updates, or monotonicity of object states in a semi-lattice), any CRDT provably converges, provided all replicas eventually receive all operations. A CRDT requires no blocking synchronization: an update can execute immediately, irrespective of network latencies, faults, or partitioning; the approach is highly scalable and implies fault-tolerance.

Notes *(collected by members of the audience)*

- **Problem:** Data is replicated, failures are common, latency is high. With software transactional memory, the approach is to restart, while in this distributed setting, such rollbacks are more difficult.
- **Solution:** Instead of conflict detection, this work performs correct conflict resolution. One technique they suggest us to use Replicated Data Types (RDTs) of which there are two kinds: convergent and commutative. These are standard objects with the restriction that all modification operations must commute, or if they do not, one has to define a conflict resolution policy (e.g., to reconcile the effect of 2 add's into a set performed at different places that now need to be merged). Hence, they need to manually define semantics of merging when there is a conflict. Another restriction on the APIs of the RDT is that the API cannot both modify the structure and return an observed result at the same time (e.g., add(k)/r interface is not allowed). A specific example of an RDT that was presented is the observe-remove Set (ORset).
  The correctness condition is defined w.r.t a particular data type (i.e., the correctness condition is specific to the data structure). For the ORset, the condition has the form "for all , there exists", which is expensive to check. The work did not present a general correctness condition and it was unclear how to obtain such a condition (say to non-commutativity). They mentioned that they performed dynamic test generation of replicated data types and have done some work in formalizing the different data type implementations.
- Large scale sharing involves data replication which is easy for immutable data but hard for mutable data. Assume: distributed, large-scale, heterogeneous, partial replication, high latency, failures, ... Conflict detection or prevention does not scale – need conflict resolution. Could use only commutative/convergent data types, also called confluent, etc. Primarily data types for containers, but also things like counters and editing of a sequence – point is to achieve conflict freedom by design. Eventual consistency: replicas that have seen the same updates achieve the same state. Discussion of possible semantics for sets, and what happens when different nodes perform add and remove on the same element. Work on defining semantics based on causal history. Many interesting follow-up issues: composing these data types; What about transactions? What semantics? Dataflow programming model; partial replication; bounding divergence.
- **Q**: How generic is the approach that you proposing? The specifications are object-dependent? **A**: There are some fundamental rules that can be abstracted, but the semantics of the objects need to be taken into account.
  Q: What is the relation between linearizability and the specification of the CRDT set? **A**: There are some similarities in the definition but it is clearly not equivalent.
  Q[Michael L. Scott]: It seems that the way you define the semantics for dealing with concurrent updates, affect the actual probability of serializing updates? **A**: This is actually true, but it cannot really be avoided.
  Q: What is the state of the art for checking the correctness of CRDTs? **A**: Some work on static checking, not dynamic. The problem of dynamic testing in distributed settings is actually quite complicated. The problem here does not appear to be significantly different.
  Q: What are the overheads that you need to pay? It seems that you need to transmit a large amount of information capturing chains of updates applied by all replicas among which interactions have occurred. **A**: This can be an issue in fact. There are however solutions that try to address precisely this issue, e.g., (dotted) version vectors.

4.5 Remaining foundational issues for thread semantics

Hans-J. Boehm (Google – Palo Alto, US)

Shared memory parallel machines have existed since the 1960s and programming them has become increasingly important. Nonetheless, some fairly basic questions about parallel program semantics have not been addressed until quite recently. Multi-threaded programming languages are on much more solid ground than they were as recently as a decade ago. However a number of foundational issues have turned out to be surprisingly subtle and resistant to easy solutions. We briefly look at a few such issues focusing on finalization in Java, and on issues related to detached threads and std::async in C++.

Notes *(collected by members of the audience)*

- Specifications for multi-threaded languages have improved but there are remaining problems: Out-of-thin-air (OoTA), managed languages and finalization, and C++ detached threads and object destruction.
- Java finalization is problematic. The method finalize() runs after object found unreachable by GC. Java.lang.ref helps but does not fix everything. Only way to work around absence of finalization is to reimplement GC in user code. Problem comes up for example in mixed language case, where Java finalizer frees corresponding C++ object: Finalizable object can be collected while method operating on object is still running, but "this" pointer is no longer live resulting in call to native method on native pointer field with dangling pointer. Various dubious and awkward solutions; synchronized (this) prevents compiler from eliminating dead references. These don't seem viable. Possible solution: "KeepAlive"-decoration. Alternative solution: annotation that prevents compiler elimination of dead references to the type; current favored solution; Rule: annotate if field is invalidated by finalization or reference queue processing.
- Issues of detached threads (= a thread that can no longer be waited for by joining it) and object destruction. Thread is no longer joinable, so resources could be reclaimed when thread terminates. Problem: there is almost no way to guarantee that a detached thread finished before objects it needs are destroyed. No way to guarantee that a detached thread completes before objects it needs are destroyed. More of an C++ issue. Recommendation: do not use detached threads.
- (Reflects insights from many WG21 committee members.) Related issues with async and futures: accidentally detached threads. C++11/14 provides blocking futures only through async. Another issue: async(f); async(g) runs serially. Future's thread can possibly try to refer to stack allocated object which can go away. Various fixes with unintended consequences. Std::async() was a mistake, hoping to fix in C++17. For example by means of separate handles on results and underlying execution agent. (Reflects insights from many WG21 committee members.)

4.6 Parallel JavaScript in Truffle

Daniele Bonetta (Oracle Labs – Linz, AT)

In this talk I presented the support for parallel execution in the Truffle/JS JavaScript engine. Truffle/JS is a JavaScript engine developed using Truffle, a multi-language development framework based on the GraalVM Virtual Machine.

Parallel execution in Truffle/JS is enabled through a combination of compiler optimizations and a built-in runtime based on Transactional Memory. The work leads to several research questions about parallel programming models and runtimes for popular dynamic languages with none or very limited support for parallel execution such as JavaScript, Ruby, and Python.

Notes *(collected by members of the audience)*

- Truffle is a framework for writing high-performance language runtimes in Java. Truffle separates the language implementation from the optimizing system. So, for JavaScript, the language runtime is an AST interpreter using the Truffle API. The interface between the language (e.g., JavaScript) and Truffle is AST nodes and compiler directives: the language implementer has to write an AST interpreter in Java, using the API provided by Truffle. On ordinary Java VMs the language runtime will be executed as a regular Java application. When run using the Graal VM, the AST interpreter will benefit from automatic partial evaluation and improved performance. Wide range of languages are already implemented in Truffle (e.g., JavaScript, Ruby, R, Python). The AST interpreter self-optimizes its nodes to improve them, for instance by determining and propagating type information, inserting appropriate guard nodes, profiles, assumptions (create, check and invalidate, ...). Correct usage of the Truffle API is responsibility of the language implementer. The Graal VM takes care of automatic compilation, de-optimization to interpreter, and re-compilation of the ASTs.
- The JavaScript implementation is quite solid: runs all the ECMAScript 5 standard tests and has increasing support for ECMA 6. It also supports many extensions, including most of Node.JS (i.e., JavaScript for server-side code). Performance of Truffle/JS is comparable to V8.
- Parallelism in JavaScript is important because of the language's popularity. However, the language is single-threaded, and developers are not familiar with threading and synchronization primitives such as locks. We take a simple approach: exposing parallelism via an API with sync/async patterns. The API should be safe (i.e., semantics same as single-threaded JavaScript) and the runtime implementation should be fast (i.e., never slower than sequential). In particular, it should do well on read-dominated, functional, or scope-local workloads. The operation "map" is a simple example for such API.
- We use Truffle to enable parallelization of JavaScript functions by adding to their AST specific synchronization barriers. In this way, dynamic conflict checks are used to back out to sequential implementation. Functions can also be executed in SW transactions to resolve potential conflicts. In this case, the runtime initially assumes that all accesses are read-only or local to a transaction, and Truffle produces optimized code for this case. Guards are used to check when the workload is not read-only or tx-local.

- There is some interesting related work concerning dynamic language runtimes that shares some aspects with our approach. Examples are ASM.JS, PyPy STM, Concurrent Ruby (with STM), and RiverTrail.
- There also are some open **Q**: How can we improve best effort performance of our runtime? How can we generalize the run-time to work with other languages? What VM-level concurrency mechanisms do we need in such a multi-language scenario?

4.7 Robust abstractions for replicated shared state

Sebastian Burckhardt (Microsoft Corp. – Redmond, US)

Joint work of Burckhardt, Sebastian; Leijen, Daan; Fahndrich, Manuel

Concurrent programming relies on a shared-memory abstraction that does not perform well in distributed systems where communication is slow or sporadic and failures are likely (such as for geo-replicated storage, or for mobile apps that access shared state in the cloud). Asynchronous update propagation (a.k.a. eventual consistency) is better suited for those situations, but is challenging for developers because it requires dealing with weak consistency and conflict resolution. In this talk I explain GSP (global sequence protocol), a simple operational model for shared data using asynchronous update propagation. GSP is similar in name and mechanism to the TSO memory model, but is suitable for use in a distributed system where communication and nodes may fail.

GSP supports synchronization primitives sufficient for on-demand strong consistency and update transactions. Moreover, GSP is expressive: all replicated data types and conflict resolution policies we know of (including OT, operational transformations) can be layered on top of it.

Notes *(collected by members of the audience)*

- **Problem:** Client-cloud shared storage where one has persistence, replication and failure. Difficult to program distributed applications in this setting yet used in say mobile computing (e.g., TouchDevelop).
- **Solution:** Adopt a replicated shared state model (client has virtual copy of entire state): easier to program against than say message passing, but then one needs to relax the consistency model, due to the CAP theorem. The particular work proposes a programming model which adapts the TSO weak memory model to distributed systems (the motivation is that this model is best understood and formalized so far). The new model is called GSP: here reads are not synchronous, unlike in TSO. In GSP, there are 2 kinds of stores: confirmed and unconfirmed stores. The system propagates stores to the confirmed buffers of other processes. The runtime system also performs combining of the effects of different operations (stores) performed on the data type (e.g., add(1) + add(1) become add(2)). The approach defines types (called cloud types) which can be used to program with the GSP model and which also avoid running arbitrary server code.
- Multiple users, variety of devices, access to my workspace from all devices (and ability to run code on all these devices). Programming these things is a mess – distinguish between RAM and GUI state, persistent state, decompose into parts that run on stateless server and with persistent storage back-end.

- Lowest level: message passing (actors); next level: shared state: stateless cloud server accessing cloud storage; highest level: replicated shared state: sync when connected, etc.
- Ladder of consistency models: linearizability, sequential consistency, causal consistency, eventual consistency, quiescent consistency. Last three are about asynchronous updates. See his book on Eventual Consistency.
- How close can we get to strong consistency (earlier in the ladder)? Compare with memory models: Not a good match since memory models are for fast communication and no failures; analog of TSO perhaps?; coherent shared memory; store buffer that drains to shared memory when it can; stores asynchronous but reads synchronous; maybe more that local replicas with reliable total order broadcast?; this moves to a view of "memory" as a log of operations, and may need more general view of operations – not just read and write, but can usually be partitioned into reading and updating operations.
- Leads to model with: globally confirmed updates (a sequence), local unconfirmed updates (also a sequence). Can answer local queries from unconfirmed updates but can receive global confirmed items that precede my unconfirmed updates. State when issuing an update may be different from the state when the update takes effect. Can get used to this, but it can also bite you! May need to add various atomic operations (describe as a (pure) update; key trick, since not referring to the state).
- Invented a fixed set of Cloud Types, suitable for this kind of computing. Example: Cloud Table = ordered sequence of rows; can append at end, delete anywhere, and ask whether a row is confirmed. Can implement something like a bank account. Handles editing via entering a row describing the state change and applying a three-way merge operation.
- Reduction = a process of reducing the prefix of a log to a small state; Transactions = explicit push, pull, and confirmed property.
- **Q**: Why not using transactions? **A**: You may use transactions but this is an alternative mechanism. **Q**: There are still some anomalies that can occur in this model. If you observe the state of something in your buffer, you may observe state that can then later on be updated by a remote update.

4.8 The Adaptive Priority Queue with Elimination and Combining

Irina Calciu (Brown University, US)

Joint work of Calciu, Irina; Mendes, Hammurabi; Herlihy, Maurice
Main reference I. Calciu, H. Mendes, M. Herlihy, "The adaptive priority queue with elimination and combining," in Proc. of the 28th Int'l Symp. on Distributed Computing (DISC'14), LNCS, Vol. 8784, pp. 406–420, Springer, 2014.
URL http://dx.doi.org/10.1007/978-3-662-45174-8_28

Priority queues are fundamental abstract data structures, often used to manage limited resources in parallel programming. Several proposed parallel priority queue implementations are based on skiplists, harnessing the potential for parallelism of the add() operations. In addition, methods such as Flat Combining have been proposed to reduce contention by batching together multiple operations to be executed by a single thread. While this technique can decrease lock-switching overhead and the number of pointer changes required by the removeMin() operations in the priority queue, it can also create a sequential bottleneck and limit parallelism, especially for non-conflicting add() operations.

We describe a novel priority queue design, harnessing the scalability of parallel insertions in conjunction with the efficiency of batched removals. Moreover, we present a new elimination algorithm suitable for a priority queue, which further increases concurrency on balanced workloads with similar numbers of add() and removeMin() operations. We implement and evaluate our design using a variety of techniques including locking, atomic operations, hardware transactional memory, as well as employing adaptive heuristics given the workload.

Notes *(collected by members of the audience)*

- Review of prior techniques, report on work presented at DISC 2014. Elimination (get rid of ops that "cancel out"), delegation (server thread does work on behalf of others), and flat combining (does both, with threads taking turns as server).
- Implementation is based on skip list. removeMin is a challenge: little concurrency, flat combining good for this. The operation "add" parallelizes nicely but not so great for flat combining.
- Can we put this together and get best of both worlds? Use typical add for "large" values. Use elimination on smaller values (near removeMin active region). Small values posted to an elimination array, larger ones go straight to skip list. So, in effect have two skip lists, one for smaller values, one for larger. Must adaptively adjust the boundary, in either direction. Boundary movement is done with a reader/writer lock.
- Better scalability than previous methods. If removeMin not as common, scalability can suffer, apparently because of RW lock.
- What about using HTM on that? Simplistic approach has too many conflicts. But when done sensibly (put all CAS for a given add into a single transaction), obtains better speedup. On 8-thread Haswell machine, get maybe 30% better throughput.

4.9 Concurrency Restriction Via Locks

Dave Dice (Oracle Corporation – Burlington, US)

URL https://blogs.oracle.com/dave/resource/Dagstuhl-2015-Dice-ConcurrencyRestriction-Abr.pdf

As multi-/many-core applications mature, we now face situations where we have too many threads for the hardware resources available. This can been seen in component-based applications with thread pools, for instance. Often, such components have contended locks. This talk shows how we can leverage such locks to restrict the number of threads in circulation in order to reduce destructive interference in last-last caches, as well as in other shared resources.

Notes *(collected by members of the audience)*

- Scalability collapse (often because of locks). Difficult to choose best number of threads since adding more degrades performance after a certain point.
- Describes a synthetic benchmark demonstrating performance of various kinds of locks.
- Solution approach: constrain concurrency at any given lock.
- The scalability collapse point may be at more threads than the best-performance point!

- Improvement may come for subtle reasons, such as improved cache miss behavior in the critical section when fewer threads are running. Competition can be for a variety of different resources, even in hardware. Effects are amplified with transactions because of transaction aborts' wasted work.

4.10 Why can't we be friends? Memory models – A tale of sitting between the chairs

Stephan Diestelhorst (ARM Ltd. – Cambridge, GB)

With my relatively recent transition from a strong memory model (AMD64, similar to TSO) to a weakly ordered architecture (ARM), and some exposure to software while working in a HW company, I would like to put out some points for discussion on why useful weak HW models cannot keep their reasonable properties once they are lifted to a language level programming model without causing prohibitive amounts of fences or other ordering primitives (such bogus branches) behind every global memory access.

I think in the many-core programming world, we can make coherence stay, but I would argue that keeping a strong memory model might not be possible. Therefore, we ought to see what makes using weak memory models hard on today's and tomorrow's weakly ordered machines and fix the semantics for future architectures.

Notes *(collected by members of the audience)*

- Stephan recently moved from AMD to ARM, and thus from a strong to a weak memory model. ARM has weak ordering and no store atomicity (some processors may see a store while other do not yet). At the same time: address dependency preserves order, data dependency preserves order, and control dependency orders subsequent writes.
- The dependencies prevent (at least certain) out-of-thin-air cases, but compiler optimizations can change or remove dependencies! Naive examples violate intuition, but one can modify the examples in ways that make the intuition go away, and thus help to illustrate why the Java memory model has been so difficult to nail down. Can't afford to make every read/write "sacred", so role of compiler, language definition and CPU spec all gets complicated. (We see a 3-way dance between architecture, language, and compiler.)
- Hard to fix at any single point of language, compiler, and architecture. Architecture isn't likely to change. For ARM, "fixes" would mess with goal to be small, fast, and energy efficient. Compilers are unlikely to change either [slides skipped for time].
- Hypothesis: we need to fix the languages. But how? Force a function to implement dependence of all output on all inputs? Probably not: can't implement Haskell or R or anything else that wants to avoid fully evaluating everything. Force a fence or conditional branch after every load? Use use ARM v8 ld.acq a lot?
- **Q**[Charles E. Leiserson]: What about "observer functions"? These indicate, at every point, what write you'd see if you chose to read [Frigo & Luchangco]. They're dependency-based. They avoids anomalies where threads A and B go through a common state (and thus should have equated views) but didn't actually *look* at anything – Heisenberg-ish. **A**: Dependences serve to break "cycles of self fulfillment" in out-of-thin-air examples. **Q**[Torvald Riegel]: It's not clear you can fix everything at the language level: the compiler

wants to "change the program". **Q**[J. Eliot B. Moss]: And even if the language is "right", programmers get it "wrong". **Q**[Hans-J. Boehm]: The real problems arise when the program fails to specify enough. What are the semantics then? **A**: It would be really useful to have compelling "real world" examples – things architects would accept as "more real" than the usual "brain teaser" examples.

4.11 Application-Directed Coherence and A Case for Asynchrony (Data Races) and Performance Portability

Sandhya Dwarkadas (University of Rochester, US)

Joint work of Dwarkadas, Sandhya; Shriraman, Arrvindh; Zhao, Hongzhou
Main reference A. Shriraman, H. Zhao, S. Dwarkadas, "An application-tailored approach to hardware cache coherence," *Computer*, Special Issue on Multicore Coherence, 46(10):40–47, 2013.
URL http://dx.doi.org/10.1109/MC.2013.258

I described an application-tailored approach to supporting coherence in hardware at large core counts and present two complementary approaches to scaling a conventional hardware coherence protocol. SPACE is a directory implementation that reduces directory storage requirements by recognizing and storing only one copy of the subset of sharing patterns dynamically present at any given instant of time in an application. Protozoa is a family of coherence protocols designed to adapt the data transfer and invalidation granularity to match the spatial access granularity exhibited by the application. Compared to conventional one-size-fits-all approaches, these designs match coherence metadata needs and traffic to an application's sharing behavior, allowing an application's inherent scalability to be leveraged.

I also showed empirical data to make a case for architectures, runtimes systems, and parallel programming paradigms to allow applications to tolerate data races (operate asynchronously) where desired and to design for performance portability. I discussed our efforts at the operating system, runtime, and programming paradigm level to enable automated techniques incorporating both application and hardware knowledge of sharing and memory access behavior for resource-commensurate performance.

Notes *(collected by members of the audience)*

- The talk is concerned with different ways of implementing coherence. Whether it is done in SW or HW is, at least in principle, irrelevant. User-defined consistency can be useful. Distributed domains often do not require strong consistency.
- Scalability enhanced by metadata storage compression. Conventional Full Map Directory is rather large as you need 1 bit per cache line. 1 bit per processor per line, 64 byte line, 128 cores, means directory is 1/4 of shared cache size. Compression: multiple blocks may have same sharing pattern. Compress the information in application-specific ways. Limited number of sharing patterns allow compression in a directory Decouple sharing patterns from cache blocks for smaller directory. Experiments on benchmarks shaw very high rates of sharing. Uses only patterns found in the applications. Sharing PAttern-based CoherncE (SPACE): n to log n bits to describe sharing. Sublinear scaling. 57 % area, 50 times energy cost at 16 cores. Can be a lot of waste in cache lines – perhaps as low as 21% of a 64 byte line is accessed. An alternative to SPACE are "shadow tags" that are similarly compressed but are more energy costly. Want adaptive granularity.

- Communication efficiency. Key to a good communication efficiency is granularity control. The Protozoa adaptive coherence granularity protocol reduces the communication demand significantly. Adapt coherence traffic to sharing behavior. Eliminate read-write and write-write false sharing. Metadata storage comparable to conventional schemes. Reduces on-chip traffic by 26%, traffic 37% across 28 benchmarks. Overhead is a function of application behavior.
- Case for data races. Some applications converge despite data races. Asynchronous algorithms. Successive over-relaxation, SVM. Converges even in the presence of data races. Here the case is being made that the compiler needs to allow for having racing reads and writes without enforcing consistency. This enables noticeable speedups. Synchronization needed to know whether data is current round or previous can be eliminated.
- Genome analysis: When looking at clusters of multi- or many-cores, overall performance very much relies on locality as can be enforced through pinning. Need to allow programmer to say this and have it carried out efficiently. Can detect sharing and adjust things in the OS to improve scalability. Performance portability remains a challenge. Proposed Linux runtime monitor detects sharing in applications.

4.12 Future of Hardware Transactional Memory

Maurice Herlihy (Brown University, US)

Maurice Herlihy led a discussion of the future of hardware transactional memory (HTM) focused around three questions.

First, are progress guarantees a prerequisite for widespread adaptation of HTM? Opinion was divided: some felt that such guarantees were necessary, but many felt that lock elision provided enough of an alternative that stronger guarantees were not necessary.

Second, is the ability to issue non-speculative instructions from a hardware transaction essential to constructing hybrid schemes that combine hardware and software? Here, the opinion was mixed. IBM's Power architecture provides the ability to suspend a transaction to execute a limited set of non-transactional operations, but there was some question whether that mechanism was to inefficient to use.

Third, there was a broad consensus that lock elision was an effective technique, but only if the application programmer could control the retry policy, implying the Haswell's built-in lock elision mechanism was too inflexible.

Finally, there was widespread agreement that better debugging support was needed.

Notes *(collected by members of the audience)*

- Is HTM doomed without progress guarantees? Too many code paths: fast path HTM, slow path on HTM abort, slow-slow STM-only version. How would you state a guarantee? Different systems may need different design points.
- Did the Haswell bug ruin everything? Is HTM just too hard to get right? No correctness bugs reported for IBM implementations.
- Is Hybrid TM hopeless without non-transactional operations? But then what do non-transactional writes mean? (We could argue for a decade!) Immediate NT reads,

immediate NT writes, and delayed (on-commit attempt) writes are all possibilities. Maybe logging read/write sets would be better HW primitive?
- What should our HW "ask" be?
- HTM and memory management? Hazard pointers make the common case expensive (because of memory barrier at each traversal). May choose transaction size adaptively to match appropriate degree of speculation - do multiple state transitions as a single one, speculatively.
- Is lock elision unloved? Customers want to code their own retry policies ...

4.13 On verifying concurrent garbage collection for x86-TSO

Antony Hosking (Purdue University, US)

Joint work of Peter Gammie; Hosking, Antony; Kai Engelhardt

I reported on the machine-checked verification of an on-the-fly, concurrent, mark-sweep garbage collector in Isabelle/HOL. The collector is state-of-the-art in that it is designed for multi-/many-core architectures with weak memory consistency. The proof explicitly accounts for both of these features, incorporating the x86-TSO model for relaxed memory semantics on x86 multiprocessors. To our knowledge, this is the first fully machine-checked proof of such a garbage collector. We couch the proof in a framework that system implementers will find appealing, with the fundamental components of the system specified in a simple and intuitive programming language. The framework is sufficiently detailed that correspondence between abstract model and assembly coded implementation is straightforward so as to allow formal refinement from model to implementation.

Notes *(collected by members of the audience)*

- Proved essentially that garbage collector on multi-/many-core architecture doesn't collect non-garbage (i.e., correctness). Concurrent, on-the-fly mark&sweep collector that does not compact. Fragmentation tolerant (cache line size fragments). Schism CMR RTGC.
- Challenges: Concurrent system; memory is not sequentially consistent (Sewell et al.'s x86 TSO model); mutators are not data race free; model is fairly realistic; and formulating model and invariants in a manner friendly to systems people.
- Model: mutator processes, collector process, system component (handles the HW memory model). Tricolor abstraction used; marking propagates a grey wavefront across the reachable heap.
  Collector techniques: insertion barrier incremental update; deletion barrier snapshot; white allocation when not marking; black allocation when marking.
  Collector code structures: Series of initial handshakes that establish some invariants; mutators put roots into a worklist; collector processes worklist, inserting new objects as necessary; termination check phase (grab any more fodder from mutators); reclamation sweep phase.
  Marking: use a CAS to mark (and claim) an object, then add to worklist.
- Modeling x86-TSO: from Sewell et al.; Buffers writes in order; reads from the buffer; bus lock for larger atomic operations.

- Proved correctness of model of the code. Boundedness of TSO buffers was discussed. They are not. Proof has to consider reachability of pointers in TSO buffers. The roots may be in the write buffer. Snapshot ensures reachable white objects are reachable from a grey object. Write barriers insure greying, and the CAS causes an immediate effect. Model TSO via message processing to system process. Proof uses Lamport-like techniques from 70s and 80s.
- Modeling language: imperative language with message passing, CIMP. Original code turns into something very similar in CIMP.
- Invariants: Universal – only data; Local – talk explicitly about control locations ("pc values"); at_p l s – process p is at label l with state s. Push all invariants across all transitions (in practice some things end up being local invariants; can use full HOL in doing the proof).
  Constructing the invariants: Track what the mutators know about the current GC phase; order of writes to different variables mostly doesn't matter. It can slice the system and get small relations over smaller parts. Worst operation is marking (of course!). TSO subtlety: deletion barrier marks a reference that is read, so what exactly is read? Finally use tricolor invariants.
- Proof technique: monster induction over all states. Tactics eliminate the trivial cases, allowing focus on the interesting parts. Annoying thing: have to "carry" invariants across the TSO buffer to memory, i.e., invariants get stated twice, once in a local form and again universally.
- Intended to eventually also model ARM Power. Feasibility not yet clear. Lack of store atomicity on ARM is likely to complicate matters. Similar proof for C11 model possible?
- Result takes 2 CPU hours of proof. Only a safety property. Liveness not yet proven. Would like to eventually prove that all objects are eventually collected. This model does not really allow thinking about performance, only correctness.
- **Q**: Do redundant work instead of CAS when marking? **A**: Probably wouldn't require much additional work.

4.14 Efficiently detecting cross-thread dependences to enforce stronger memory models

Milind Kulkarni (Purdue University, US)

Joint work of Kulkarni, Milind; Bond, Michael; Sengupta, Aritra; Biswas, Swarnendu; Zhang, Minjia; Cao, Man; Salmi, Meisam Fathi; Huang, Jipeng
Main reference M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, J. Huang, "OCTET: Capturing and controlling cross-thread dependences efficiently," in Proc. of the 2013 ACM SIGPLAN Int'l Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA'13), pp. 693–712, ACM, 2013.
URL http://dx.doi.org/10.1145/2544173.2509519

In this talk, I discussed two results from an ongoing project.

First, I described a system called Octet, that provides detection of dependences between threads. Octet operates by associating a thread ownership state with every object. Prior to accessing an object, Octet checks the state of the object; an access incompatible with the object's state implies a potential cross-thread dependence. Octet uses an optimistic protocol for managing these ownership states: most accesses do not require state changes, and Octet

requires no synchronization for these accesses; synchronization is only required when the state must change.

Second, I described how we use Octet to enforce a stronger memory model, statically bounded region serializability. The memory model considers each thread to be a sequence of statically-bounded regions; execution appears to be some serial interleaving of these regions. We implement this memory model through a combination of Octet to provide two-phase locking (ensuring region atomicity) and compiler transformations to support region rollback to avoid deadlock.

Notes *(collected by members of the audience)*

- Safely and efficiently detecting cross-thread dependences (think: ownership tracking). At least two threads accessing an object and at least one writes to the object. Safely: time of check vs. time of update of the meta data; Efficiency: need to protect the meta data. Safe to do an atomic operation on an object's metadata before each use. But that gives a 3x slow down. **Q**[Charles E. Leiserson]: Is 3x slowdown is reasonable for debugging? **A**: It is, but if I want to do record/replay or use it for STM, I want it to be faster.
- Octet protocol: Fast path checks current ownership (meta data is already in a "good state"), and if ok, proceed without expensive synchronization operations. But if state is not what I need, there is a more complex, instrumented, bias-locking slow path (presumably rare). The slow path causes a thread to wait until the object's current owner reaches a safe point.
- Proposal is Statically Bounded Region Serializability (SBRS). Optimize heavily within a region. Static bounding is not a limitation: dynamic regions do not lead to larger regions in practice.
- Capturing of dependencies can be be used for atomicity checking, STM, record & replay. Example: implementing a stronger memory model (using Octet) using a 2PL approach and combined with some rollback mechanism. Overhead of benchmarks: 13% cost for Octet with no coordination, 26% with coordination, 30% for SBRS.
- Slide 9: **Q**[Stephan Diestelhorst]: What is the initial state of an object? **A**: Could be invalid. Could be owned by the allocator.
  Slide 11: **Q**[Michael L. Scott]: Is this like asymmetric/biased locks? **A**: Yes. Essentially, biased read/write locks on every object.
  Slide 13: **Q**[Stephan Diestelhorst]: Do you require thread 1 to wait until it gets to the safe point? **A**: Thread 1 does not have to do unconditional wait. There is a potential for deadlock. Trick: while a thread is spinning, it can give access to other threads. This scales. T1 does not need to know which object it needs to access. [J. Eliot B. Moss]: Moss: T2 must know that T1 is still the youngest. The CAS serves the function of saying "I need this object no other thread should have access to it" **Q**[Martin T. Vechev]: Can I encode arbitrary state transition, such as type state? **A**: Yes, we believe we can encode other types of state. **Q**[Martin T. Vechev]: Is the protocol specific to this automata? **A**: It is not specific to this automata. Coordinate with a share state requires you to coordinate with all other threads.
  Slide 18: **Q**[Stephan Diestelhorst]: Are function calls allowed in a region? **Q**[Jose Nelson Ameral]: Does inline changes the atomicity properties of the program? If the programmer writes a set of statements into a function expecting those to be atomic, then after the compiler inlines that function the atomicity property would be lost. **A**: You can use synchronization annotations to ensure that the atomicity is enforced.

4.15 Hardware Transactional Memory on Haswell-EP

Viktor Leis (TU München, DE)

Intel's Hardware Transactional Memory feature TSX was initially launched for Haswell desktop CPUs with 4 cores. Only recently, systems based on Intel's mid-level server platform Haswell-EP became available. Haswell-EP supports two sockets and up to 72 hardware threads. On such many-core systems, transactional memory is both more desirable and more challenging.

In this talk, I will present a number of experiments on a dual-socket Haswell-EP system with 28 cores. The results show that TSX can indeed achieve good performance and scalability on NUMA systems with many cores. However, there are a number of non-obvious pitfalls that must be avoided.

Notes *(collected by members of the audience)*

- The talk is concerned with performance evaluation of HTM in the context of data bases.
- Server-class Haswell is Intel's mid-level server platform. It comes with up to 18 cores per socket, 72 HW threads with 2 sockets, and supports TSX (must be explicitly enabled, due to "the bug").
- Experimental setup: global fallback lock using Hardware Lock Elision (HLE); Alternative: implemented by RTM (still lock elision); Workload: (a) adaptive radix trie (fanout 2-256), as for a main memory DB, (b) random lookups in 64M entry trie, (c) 64M random inserts into initially empty trie (hard to turn into a totally non-blocking data structure; typically touches 10 to 12 cache lines; should not lead to capacity issues in most cases). Measurements on Intel Xeon E5-2697 v3 on 2 sockets 14 cores each 2 HW threads each system. (One interconnect ring per 7 cores, two rings are linked, then those are linked across socket with QPI, rw_spin_lock totally does not scale, no sync does).
- A conflict-free look-up benchmark with locking test shows bad speedup with read-write spin lock. The theoretical peak is almost 100M ops/s locks bring it down to less than 25M. An atomic counter does not do much better. Built-in HLE does not speed up, but customized HTM performs much better, but sensitive to the restart policy. If you are willing to do enough restarts, get speedup almost as good as no sync. The more restarts, the better. 7 or more restarts is scalable. Why? In this case, aborts are mostly spurious, so fallback is harmful.
- For random inserts, which do have conflicts, the scalability was dominated by the memory allocation policy, with malloc the worst (no speedup), tcmalloc better (scaling stops at 28 threads), with a combination of memory pre-allocation and zeroing out doing the best.
- The NUMA behavior was tested by placing threads on 1 cluster, 1 socket, and 2 sockets. Speedups were roughly the same, even though the actual time was faster for more local setups. (Lookup: better speedup by spreading threads around, both across clusters and across sockets; Insert: same effects). Overall rate better when memory being used is restricted to unit running the threads. The more local the threads are the better the performance is; the scalability stays the same, though. HTM works over NUMA.
- Conclusions: HTM scales to NUMA, built-in HLE does not scale. Despite very few collisions, or maybe because of very few infrequent collisions, large restart numbers (>20) seem essential. Kernel traps within the transactions have deadly effects.

4.16 What the $#@! Is Parallelism? (And Why Should Anyone Care?)

Charles E. Leiserson (MIT – Cambridge, US)

Many people bandy about the notion of "parallelism," saying such things as, "This optimization makes my application more parallel," with only a hazy intuition about what they're actually saying. Others cite Amdahl's Law, which provides bounds on speedup due to parallelism, but which does not actually quantify parallelism. In this talk, I reviewed a general and precise quantification of parallelism provided by theoretical computer science, which every computer scientist and parallel programmer should know. I argued that parallel-programming models should eschew concurrency and encapsulate nondeterminism. Finally, I discussed why the impending end of Moore's Law – the economic and technological trend that the number of transistors per semiconductor chip doubles every two years – will bring new poignancy to these issues.

Notes *(collected by members of the audience)*

- Parallelism: simple model of parallel computation: DAGs. Strand is a serial chain of executed instructions. Dependency is a necessary ordering relationship. Usual notion of forks and joins in the DAG. Programming language can express these. Can schedule dynamically at run time. Amdahl's Law – it does not of itself quantify parallelism, only potential speedup. Can model the time required on P processors using the task DAG. Work Law: longest path gives min time required. Span Law: largest number at once gives max speedup. Theoretical model says super-linear speedup is not possible; in reality, other effects can sometimes produce super-linear speedup. Still, can describe max (theoretical) speedup as ratio of time required for one processor to time required by an unbounded number of processors. If you use more processors, they cannot be fully utilized. Can prove that Cilk's scheduler gets near perfect linear speedup if the parallelism substantially exceeds the number of processors available. Enables a straightforward scientific approach to speeding up your programs.
- Concurrency: Situation is much more complex. Concurrency introduces interactions between threads that often reduce available parallelism. Theoretical models not very strong. Should eschew concurrency on most programming. Need to move away from concurrency toward determinism. Concurrency essential to implementing the platform. But best done by experts, once. Historical analogy to "Goto Considered Harmful", which led to structured control constructs (arguably "complicated" things: goto is "simple" – but it was good in the end).
- Why care? Moore's Law. At 14nm now; will get down to 5nm but likely not much more (at least not economically). Limit about 2020 or 2022 according to Bob Colwell. Solution: replacement technologies can still help, but they're going to be software technologies: computer science.

4.17 Bringing concurrency to the people (or: Concurrent executions of critical sections in legacy code)

Yossi Lev (Oracle Labs., US)

In this presentation I have discussed a few of the challenges that people are likely to encounter when making critical sections in legacy code executed concurrently (e.g. using transactional memory), and provide some examples of how some of the data structure and infrastructure work we've been doing in the last few years can help addressing these challenges.

Notes *(collected by members of the audience)*

- How do we make HTM useful to as many people as possible in the near term (as well as the long term)? Need near-term benefits to motivate vendors to keep investing.
- Code in most critical sections was not designed to run concurrently! Need to avoid writing same value: turn x = v; into if (x != v) x = v; , if it is likely that x==v will hold in most cases. Clearly, we do not want all writes to become conditional, but for some writes this can be very effective, esp. for variables whose types have a small number of values – booleans, node color in RB tree, phase indicators, ...
- Minimize time period from a write to the end of the critical section – will reduce conflicts/aborts; pad data that frequently changes, to avoid false conflicts;
- Counters: shared counters in critical sections are common and are often the cause for failure of optimistic approaches (such as HW transactions). Per-thread counters may lead to bad performance if the total value (i.e., the sum of these counters) needed to be calculated frequently; per-core/node counters work better, as there is only a small set of counters to read, and this set is known upfront. In some cases the update to the counter dos not have to happen atomically with the rest of the critical section operation, in which case a separate fetch-and-add outside of the critical section, can help. Can sometime also use a solution that increments counters probabilistically, if approximation will do [1]. Finally, sometimes you do not have to know the exact value of the counter, but simply some property of it – e.g., use SNZI: Scalable Non-Zero Indicator, to only know if it is 0 versus not-0. SNZI works quite well with HTM.
- **Q**[Hans-J. Boehm]: May want to get compilers to do many of these optimizations. Also need to prevent compiler from undoing them if you've done them by hand. **A**: Declaring variables "volatile" helps prevent the compiler from undoing, but at the longer term we want compilers to optimize code that is executed inside a transaction differently.

References

1 Dave Dice, Yossi Lev, and Mark Moir. Scalable statistics counters. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'13, pages 43–52, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1572-2. DOI 10.1145/2486159.2486182.

4.18 Towards Automated Concurrent Memory Reclamation

Alexander Matveev (MIT – Cambridge, US)

Joint work of Alistarh, Dan ; Eugster, Patrick; Herlihy, Maurice; Leiserson, William M.; Matveev, Alexander; Shavit, Nir
Main reference D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, N. Shavit, "StackTrack: An automated transactional approach to concurrent memory reclamation," in Proc. of the 9th European Conf. on Computer Systems (EuroSys'14), pp. 25:1–25:14, ACM, 2014.
URL http://dx.doi.org/10.1145/2592798.2592808

The concurrent memory reclamation problem is that of devising techniques to allow a deallocating thread to verify that no other concurrent threads, in particular ones executing read-only traversals, have a reference to the block being deallocated. To date, there is no satisfactory solution to this problem: existing tracking methods like hazard pointers, reference counters, or epoch-based RCU, are either prohibitively expensive or require significant programming expertise, to the extent that using them is worthy of a conference paper. None of the existing techniques are automatic or even semi-automated.

This research project will take a new approach to concurrent memory reclamation: instead of manually tracking access to memory locations as done in techniques like hazard pointers, or restricting accesses to specific methods as in RCU, we plan to use the operating system and modern hardware's transactional memory tracking mechanisms to devise ways to automatically detect which memory locations are being accessed, and allow accesses in any point in the code. This will allow to design and implement a new class of automated concurrent memory reclamation frameworks, making them relatively easy to use and allowing them to scale, so that the design of such structures can become more accessible to the non-expert programmer.

Notes *(collected by members of the audience)*

- Consider case of logical-then-physical deletion in a (concurrent) linked list. Hand-over-hand locking has too much overhead therefore use unsynchronized traversals. But that complicates memory reclamation. How do we know when a node can be reclaimed? Need to track both thread-local (stack) references and global references, e.g., passed through task queues in the heap.
- Current solutions: reference counting, hazard pointers [Maged Michael, Herlihy et al., Braginsky et al.] or epoch/RCU-based.
- Traditional approaches worry only about the thread-local case. "Extended memory reclamation" addresses the global exchange references as well – things that point to nodes that have been removed from a shared abstraction but are not gone from the system (and might be added back in), and thus should not be reclaimed. Need to distinguish "permanent" references between node of a data structure.
- Stack Track system uses HTM to scan thread stacks for transient references. Automatically adapt and split these into smaller transactions when capacity aborts are detected. If this is not an option (no HTM, or too size-restricted), can emulate in SW. Do stack scan of a thread to find interesting pointers. Software StackTrack as fallback for HTM StackTrack.
- Doesn't have a general solution yet for global exchange variables. Currently using a visible pool of global references. Change mappings on pages to prevent concurrent changes.

4.19 Portability Issues in Hardware Transactional Memory Implementations

Maged M. Michael (IBM TJ Watson Research Center – Yorktown Heights, US)

Recent hardware transactional memory implementations that became commercially available in recent years have differences in their architectural and performance features. These differences can lead to programming pitfalls and raise functional and performance portability issues. There is a risk that HTM users learn the wrong lessons about HTM in this early stage of its commercial availability, and influence future HTM designs and uses based on such lessons. In this talk, I discussed differences among HTM implementations and potential pitfalls.

Notes *(collected by members of the audience)*

- This talk presents an overview of current HTM designs and discusses interfacing issues. Differences between main architectures.
- Z: has constrained transactions that restrict the operations that can occur within a transaction – the other systems don't. An obstruction-free transaction. If you follow certain constraints, then in the absence of conflicts, the transaction will eventually complete. No need for failure handler. Not portable to Intel or Power8. In effect, really small txns.
- Haswell: has built-in hardware lock elision – the others don't. Backwards compatible. Library may not have a standard "I'm free" value. Hazard: reading lock bit in critical section can see unexpected unlocked value. RTM does not impose many unique constraints on coding.
- Power8: Power8 has suspend/resume. Allows even system calls within the transactions. Suspends current transaction, allows non-transactional execution until resume. Some restrictions: while suspended, cannot write to memory read by transactions. Can check for transaction failure while suspended, but handlers called only on resume. Non-transactional loads good for loop parallelization, hybrid TM, reducing read set. Non-transactional stores good for debugging, conflict resolution, must be used with care. Read and write sets can be explicitly controlled through switches between suspend and resume states. Resolution is explicit. Nested entry primitives can lead to stack corruptions upon abort. This requires explicit programming support.
- (Non-)Portability examples: (a) TLE does not work well with indiscriminate non-transactional stores (e.g., to thread stack). Start in function, then return before the end of the transaction, pop stack, write to stack frame non-speculatively. If the transaction fails, we end up with a corrupted stack. Z supports non-transactional stores. Invisible to other threads until the transaction ends. They become visible even if the transaction aborts. Useful for debugging, not for communication. (b) Power8 Rollback only transactions. Single-thread speculation, no shared data, no conflict detection, no order guarantees. Rollback Only Transactions does not do any conflict detection; that are intended for single threaded speculation only. However, they can be interleaved with regular transactions. In that case they are treated as atomics. (c) XEND/TEND outside of transactions: some processors fault, others don't, so not convenient for portability. Cannot call Haswell XEND outside transaction (must check first), but ok to call TEND

in Z and Power8. (d) Big variation in capacity. Encourages different programming styles. (e) HTM caching policies also important and different across systems, but hidden from programmers. (Handling for Power8 depends on the mode but it typically bypasses L1.) (f) Variation in overhead of using transactions (compared with atomic ops). Depending on the application circumstances, the overhead can easily be around 50%. Different conditions on different systems.

4.20 Local Combining on Demand

Erez Petrank (Technion – Haifa, IL)

Main reference D. Drachsler-Cohen, E. Petrank, "LCD: Local Combining on Demand," in Proc. of the 18th Int'l Conf. on Principles of Distributed Systems (OPODIS'14), LNCS, Vol. 8878, pp. 355–371, Springer, 2014.
URL http://dx.doi.org/10.1007/978-3-319-14472-6_24

Combining methods are highly effective for implementing concurrent queues and stacks. These data structures induce a heavy competition on one or two contention points. However, it was not known whether combining methods could be made effective for parallel scalable data structures that do not have a small number of contention points. In this paper, we introduce local combining on-demand, a new combining method for highly parallel data structures. The main idea is to apply combining locally for resources on which threads contend. We demonstrate the use of local combining on-demand on the common linked-list data structure. Measurements show that the obtained linked-list induces a low overhead when contention is low and outperforms other known implementations by up to 40% when contention is high.

Notes *(collected by members of the audience)*

- Lock Combining = Threads help each other instead of contention on a single lock. Combining waiting pushes and pops: one thread grabs the lock then does all the pending operations. Known to speed up stacks and queues.
- Lock combining for sorted linked list implementation of set. Effective combing when there is contention: elimination of inserts and removing of duplications (same key). Locks on individual nodes, but designed so that contains and search do not need to lock. Apply combining on contended locks: holder does all operations that queue on the same lock; can also eliminate complementary operations (interestingly, regardless of order (because order isn't really defined)). When a waiter wakes up, may need to check the situation; the combiner could also wake him up earlier, telling him that he needs a different lock. Doing this on demand, i.e., only when there is contention. Don't do combining if you get the lock immediately. Combining is local: happens only on the contended lock.
- What about operations that require multiple locks (such as remove)? Split into separate sub-operations, each acquiring one lock. Note that first lock remains held while second lock is acquired and its sub-operation done. Need to preserve serializability. Preserve two phase locking.
- Have integrated this with reentrant Java locks [Doug Lea]: use the waiting-thread list of the lock instead of duplicating the wait queue.

- Performance comparison with other implementations of same data structure on machine with 64 HW threads; lock-free list eventually wins at high contention, lock combining list is competitive or dominant in lower/moderate contention situation. Lock-free does better for high thread counts. Better than combining with single global lock.
- [Yossi Lev]: Skip lists would be an interesting candidate. [Martin T. Vechev]: There are variants that don't use the deleted bit. May be easier.

4.21 Current GCC Support for Parallelism & Concurrency

Torvald Riegel (Red Hat GmbH – Grassbrun, DE)

I gave an overview of the new features related to parallelism and concurrency in the upcoming release 5 of the GNU Compiler Collection.

Notes *(collected by members of the audience)*

- Presentation refers to GCC 5 which is in stabilization phase, get from SVN.
- Parallelism in C++: OpenMP4 support (including offload to Xeon Phi and Nvidia PTX back end), OpenACC, and Cilk Plus ([Charles E. Leiserson]: metadata is missing. Intel mostly working on it.)
- C/C++ memory model fairly well supported. C++11 memory model is complete, frontend parses everything; testing of the front-end is done.
- TM support in C++ experimental. Older version of the TM for C++ spec is implemented. _transaction_atomic = atomic_commit and _transaction_relaxed = synchronized. Some additional attributes for the tm-safety annotations; additional control for bypassing instrumentation, and manually specifying both versions directly; new feature: multiple/different code paths for instrumented and non-instrumented code, possible to plug-in custom libraries.
- TM Runtime library libitm supporting different STMs and HTM for a few architectures. STM: various algorithms, running on most ISAs, including ARM, Aarch64. HTM version for Power8, s390, and Intel HTM.
- **Q**: Any users of this out there? **A**: Not aware of users outside of experimentation or research; but that does not mean there are none. **Q**: What is transaction safe, list of functions from the standard library? **A**: ISO C++ study group 5 members are going through the API, marking things as safe. Problem: Claiming functions to be transaction_safe might restrict future implementations.

4.22 Forward progress requirements for C++

Torvald Riegel (Red Hat GmbH – Grassbrun, DE)

I presented forward progress requirements for C++ implementations that I have proposed to the ISO C++ committee. These requirements define what progress means in C++ and

what the differences are between, for example, OS threads and parallel tasks running on a bounded thread pool.

Notes *(collected by members of the audience)*

- "Execution agents" (EA) proposed for a Technical Specification (TS) (potential inclusion in a future versions of C++). EAs are threads of execution with different execution properties (e.g., light-weight threads, OS threads, etc.).
- Problem: Current spec reads "every unblocked process eventually makes progress". But it is open what "progress" and "unblocked" mean.
- EAs needed to talk about and specify forward progress, but with potentially weaker guarantees than OS thread. Talk describes EAs on an abstract level. Classes of progress: concurrent, parallel, and weakly parallel. Bootstrapping progress: "boost-blocking".
- C++ semantics defined in terms of an abstract machine. As-if rule = must act observably the same as the abstract machine. Progress defined in terms of steps, resulting in termination, access/change to a volatile object, or sync/atomic operation. Progress means executing a step. Also delimits what compiler writers can elide. Blocking operations and IO may be conceptually implemented as busy-waiting on a condition. **Q**[Michael L. Scott]: When writing while(); (infinite loop), would that make progress? **A**: undefined behavior.
- Flavors of EAs, i.e., classes of progress: Concurrent = every EA will progress; Parallel = every EA will progress once it has executed its first step (this allows bounded thread pools as implementation); Weakly parallel = no guarantee, but see boost-blocking (this allows non-preemptive execution and lock-step execution (such as SIMD)).
- Boosting progress: form groups of agents, if P uses boost-blocking to wait on a group of agents G, then agents in G will have at least one boosted to P's level of guarantee, if it is higher. Boost blocking vaguely like priority inheritance with transitivity.
  Example implementation: Concurrent EA = one OS thread for each EA plus round-robin OS scheduler.

4.23 Self-tuning Hardware Transactional Memory

Paolo Romano (INESC-ID – Lisboa, PT)

Joint work of Romano, Paolo; Diegues, Nuno
Main reference N. Diegues, P. Romano, "Self-tuning Intel transactional synchronization extensions," in Proc. of the 11th Int'l Conf. on Autonomic Computing (ICAC'14), pp. 209–219, USENIX Association, 2014.
URL https://www.usenix.org/conference/icac14/technical-sessions/presentation/diegues

Efficiency of best-effort HTM (like Intel TSX) is strongly dependent on the efficiency of the policies used to regulate the usage of software the fall-back path.

In this talk, I will first present experimental data highlighting the relevance of designing self-tuning mechanisms aimed to dynamically adapt the HTM fall-back policy. Then I will discuss recent and ongoing work aimed at pursuing this goal by exploiting lightweight on-line reinforcement learning algorithms.

Notes *(collected by members of the audience)*

- How many retries until going to fall back to lock/STM? How to cope with capacity aborts? Does capacity abort count as just one retry? Could: give up (drop all retries left), half (drop half of retries left), or stubborn (reduce retry count by one). How to implement fall back synchronization: single global lock, none (retry), aux (serialize on an auxiliary lock). How well does static tuning work? Compared a heuristic (as suggested by Intel) and gcc.
- There is room for improvement over these two policies. No single policy dominates. Results vary with benchmark and number of threads. Not all the optimization dimensions are relevant: it turns out that wait and aux are similar and none is rarely better (and not by much), so that dimension can be dropped. One size doesn't fit all.
- Adaptive self-tuning approach needed. How should parameters be learned, off-line or on-line? On-line seems reasonably feasible (affordable cost). Chose particular lightweight reinforcement learning methods: upper confidence bounds (for capacity aborts) and gradient descent (for number of retries in HW). At what granularity should we adapt? Per-thread and atomic block, or whole application?
  What metrics should we optimize for? Performance, power, or combination? Are they correlated? On average 0.81 correlation. But much stronger between optimal configuration for each target: 0.98. So go for time (since easier and cheaper to measure than energy).
- Two tuners (fine and coarse grain): one per thread per Atomic block, the other global. Integrated into gcc. Speedups relative to single-threaded, non-instrumented. Auto-tune works well (speedup around 4 for 8 threads for both SG and NoRec), the gains largely outweigh exploration cost. Gradient descent can get stuck in local maxima. Use random jumps to get unstuck. Sometimes it pays to rerun transaction on capacity abort.

4.24 How Vague Should a Program be?

Sven-Bodo Scholz (Heriot-Watt University Edinburgh, GB)

For many well researched problems there exist several alternative algorithms that compute solutions. Which alternative is best suited does not only depend on the overall goal but it typically also depends on many other factors, such as the actual data, the executing hardware or the way the algorithm is actually mapped onto that hardware. The choices that need to be made in this context are always a mix of decisions made by the programmer and decisions made by the tool chains that are being used.

In particular in the light of the ubiquitous availability of increasingly heterogeneous many-core systems implementation choices do not only become more complex, but the impact of the choices made are also becoming much more pronounced. With programmer productivity in mind it seems that shifting the decision making progress towards increasingly sophisticated tool chains is the only economically viable way to go. Although a lot of progress in that direction has been achieved over the last few decades, pushing this agenda further raises many rather fundamental questions such as (a) If our programs provide increasing freedom to the tool chain to adjust the programs for parallel execution, *what* is the notion of an algorithm? (b) Is it enough to specify one algorithm as a problem solution; or should we provide several a la peta-bricks? (c) What happens with determinism or provable properties

if we allow for more than one alternative? (d) Does a discussion about complexities still make sense? (e) Do we have mechanisms that allow tool chains to choose the "best" hardware to execute on?

Notes *(collected by members of the audience)*

- Problem: too many programming paradigms for parallelism; huge challenge for scientific practitioners. Desire: raise the level of abstraction. Particularly challenging in the multi-/many-core situation (consider programming GPGPUs)
- Single-Assignment C = like C without pointers and with N-dimensional arrays. Declarative/functional programming, backed with aggressive compiler optimizations. Map to lambda calculus. Tools to generate it from a variety of front-end languages, including things like MatLab.
- Pure functional intermediate form reveals many opportunities for high performance parallelism optimizations. In particular, allows major restructuring for different platforms without having to restructure source code. Allows more flexible choice of what gets translated onto special multi-cores (such as GPGPUs).
- Significant empirical evidence of viability of this approach. Can even beat hand-written CUDA code – e.g., on Anisotropic Diffusion image processing benchmark.
- Difficulty of comparing approaches. Could have multiple implementations – multiple algorithms – from which to choose. But then, what is the "algorithm"? How do we argue correctness when what's going on under the hood can vary so much? C compilers transform what is actually executed - arguably "same algorithm", but blocking, etc., are substantial transformations. But to do well on multi-/many-core versus single-thread, you need a "different" algorithm. Choice of algorithm depends on many things, most of which are not statically determined – so needs to be determined dynamically and perhaps not even deterministically. How then do we reason about correctness? Or complexity?

4.25 Persistent Memory Ordering

Michael Swift (University of Wisconsin – Madison, US)

Non-volatile memory (NVM) technologies, such as phase-change memory, memristors, spin-transfer torque MRAM, and others promise high-bandwidth, low-latency persistent storage through the standard memory interface. However, making memory persistent poses a number of challenges, including how to ensure data is durable in the presence of processor caches, and how to ensure consistency of updates.

A key challenge in persistent memory is that data residing in caches is not durable; it must be written back to NVM first. When to do this and how to order writes back to NVM are an open research question.

I discussed several models of persistent memory ordering and then raised some open questions.

Notes *(collected by members of the audience)*

- Persistent memory is a hot topic (memristors, etc.) Many research questions regarding how to enforce atomicity and consistency? This talk focuses on consistency. What is different to non-persistent memory? We need to have a commit record that allows us to recover after a crash. Why is this a problem? Write back cache: commit log is not written out in order. There is a volatile memory ordering that is defined in terms of views of CPUs. More interesting for consistency is the order at the DRAM. In the presence of persistence, state can be uncertain after a crash. Without ordering, we cannot enforce that the "commit" record be updated strictly after the other updates. **Q**[Milind Kulkarni]: Do you need to make sure that the cache is flushed in some manner? **A**: You write a record after flashing the cache to record what you have written.
- Simple (but expensive) solutions: disable caching for logs; write through cache, flush entire cache at commit. Other approaches: Mnemosyne, BPFS/epochs, and Intel's new instructions.
  [Stephan Diestelhorst]: There is a misconception that if it is out of the cache it is durable, but there are lots of buffers between cache and memory, see PCOMMIT in Intel's HW Support.
- Mnemosyne: Goal to not use any new instructions or special hardware. Primitives: ordered writes with non-cached stores (they can bypass cache, or force a line to be flushed) or using flush/fence instructions MOVNTQ/CLFLUSH. Transactions (with durability) based on tinySTM. Note: undo logging approach not great for this situation.
- BPFS/epoch barriers: An epoch is a group of writes that are delimited by a new form of barrier. Cache tracks epoch id. Epochs have to be written by the processor in epoch number order, older epochs need to be written before newer epochs. Also need to handle cross-CPU dependences. The Problem is that one does not know when the data has become durable. No way to force durability (except perhaps by force a line to be flushed). The idea is to get ordering by accessing persistent data written by a different processor. **Q**[Michael L. Scott]: Is this for current processors (EPOCHS)? **A**: No, it is for new hardware.
- HW Support by Intel (Spec from last August): CLFUSHOPT: an unordered flush; CLWB: writes back modified data but data stays in the cache (as unmodified), i.e., without flushing; PCOMMIT: commits data queued in the memory system to persistent memory, lets you know that persistent data are now durable in memory; need to use SFENCEs between these (since they're unordered).
- Generalizing persistent order: Memory Persistence: [Pelley, Chen, and Wenisch (Univ. of Mich.)]; Recovery observer model: defines order of visibility at non-volatile memory (not caches); Persist order: orders writes to non-volatile memory.
- Epochs to Strand persistence. Epochs require fitting everything into a linear order. Instead Strand associates writes with strand numbers. Strands are not ordered. Programmer or compiler can introduce explicit order strands.
- Open questions: Are there more efficient HW mechanisms (than epochs, say)? What HW mechanisms do we need for efficiently enforcing ordering? How can stores be ordered across cores (distributed transactions)? Do we need arbitrary dependence graphs? What granularity do we want for writes, e.g., a cache line? What is the appropriate programmer API? Library (key/value or object store)? Load/store? Transactions?

4.26 NumaGiC: a garbage collector for NUMA machines

Gael Thomas (Télécom & Management SudParis – Evry, FR)

Joint work of Gidra, Lokesh; Thomas, Gael; Sopena, Julien; Shapiro, Marc; Nguyen, Nhan

When running on contemporary cache-coherent Non-Uniform Memory Access (ccNUMA) architectures, applications with a large memory footprint suffer from a large garbage collector (GC) overhead. As the GC scans the reference graph, it makes many remote memory accesses, saturating the interconnect between memory nodes. This talk presents NumaGiC, a GC that addresses this problem with a mostly-distributed design. In order to maximize memory access locality during collection, a GC thread avoids accessing a different memory node, instead notifying a remote GC thread with a message; nonetheless, NumaGiC avoids the drawbacks of a pure distributed design, which tends to decrease parallelism. On Spark and Neo4j, two industry-strength analytics, with heap sizes ranging from 160 GB to 350 GB, and on SPECjbb2013 and SPECjbb2005, NumaGiC increases the performance of the collector by up to 5.4x over Parallel Scavenge, the default throughput-oriented collector of Hotspot, which translates into an overall performance improvement by up to 94%.

Notes *(collected by members of the audience)*

- Problem: large multi-/many-cores have lots of computing power but it is hard to build a GC that scales. Scaling is limited by data analytics and NUMA. Collector forces accesses to remote memories and parallel collection ends up saturating the interconnect because of the cache coherence protocol.
- Idea: Use messages. Observation: a thread mostly accesses objects it has allocated, i.e., threads mainly access local memory. Send message to the GC thread of the node to maximize locality. Requires cross-node references to be relatively rare. Heuristic: keep objects allocated by a given node on that node. But although this avoids remote memory accesses, being so strict degrades collector parallelism. **Q**[J. Eliot B. Moss]: There tends to be locality to the object reference. **A**: Sending a message is more costly than accessing one remote object. **Q**: Is this something that you tried and did not work well? **A**: Yes, for all the benchmarks, including DaCapo, etc. **Q**[Michael L. Scott]: You describe the problem as a bandwidth problem, but the solution may also affect the latency. **A**: Scalability problem comes from the saturation of the network. [Michael L. Scott]: The critical path length of the GC could be smaller with your approach even with infinite bandwidth. [Stephan Diestelhorst]: Could you prefetch to pull the object and have the same benefit in the end?
- Adaptive algorithm: Local mode: send messages when not idling; Thief mode: grab objects (pull to your node) when idling.
- Results: GC throughput 2–5x better (with heap size 3–4 x live size); application speedup of 12–66%; GC shows good scaling; memory access locality very important to GC performance. **Q**[Michael L. Scott]: AMD and Intel are both TSO. Are we reaching a point architecturally where we can see a difference in memory order between scalability of the machines. Will there be a difference in scalability between ARM and POWER chips? **A**: I do not know.

4.27 Utilizing task-based dataflow programming models for HPC fault-tolerance

Osman Ünsal (Barcelona Supercomputing Center, ES)

In this talk, I argued that task-based programming models are a good substrate to build fault-tolerant frameworks for High Performance Computing (HPC) systems.

In particular, I further advocated the use of dataflow runtimes for resilience. These runtimes facilitate fault isolation, minimize fault propagation, and help failure root-cause analysis.

I provided examples showing how leveraging task-based dataflow PMs could lead to efficient asynchronous checkpoint/restart and selective replication implementations.

Notes *(collected by members of the audience)*

- Application resilience, for instance, in the domain of climate change predictions, is an increasing concern. This is due to larger circuits, complex substrates, and complex software. MTBF on the order of tens of minutes without applying more techniques.
- Adopt a task-based dataflow programming model where task runs when all dependencies are satisfied. Envisioned for coarse grained tasks. The runtime system checkpoints the data at the start of a task as it knows the data inputs of each task. If failure occurs, the task is re-ran (as its effects are local); scales well with fault rate. Clarifies where checkpoints can be taken. Likewise, recovery tends to be fairly local. The approach uses Software CRC for error-correction, can exploit existing Intel instructions to reduce overhead. Protects only application tasks, not run-time system or OS. Task redundancy (pairs compare output; on difference, run third and take majority vote; do this only on tasks that are more likely to experience errors (based on their memory size)). Hard to handle global shared state. The approach could potentially benefit from non-volatile memory.
- **Q**: Does non.volatile memory help to achieve fault-tolerance with data flow programs? **A**: It can help to perform selective checkpointing more efficiently, but the issues do not seem different.

4.28 Commutativity Race Detection

Martin T. Vechev (ETH Zürich, CH)

Main reference D. Dimitrov, V. Raychev, M. Vechev, E. Koskinen, "Commutativity race detection," in Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'14), pp. 305–315, ACM, 2014.
URL http://dx.doi.org/10.1145/2666356.2594322

In this talk, I introduced the concept of a commutativity race. A commutativity race occurs when two method invocations happen concurrently yet they do not commute. Commutativity races are an elegant concept enabling reasoning about concurrent interaction at the library interface and generalize classic data races. I also discussed a way to dynamically detect

commutativity races based on a technique which combines vector clocks with commutativity information.

By generalizing classic read-write race detection, the work leads to many new interesting research questions at the intersection of program analysis and distributed computing. These questions are of both theoretical and practical importance. In particular, I discussed several open directions and in-progress results including: impossibility of simulating race detectors, discovering logical fragments for capturing commutativity, and black box learning.

Notes *(collected by members of the audience)*

- Commutativity race = 2 high-level (atomic) operations that do not commute and are not ordered. Useful for debugging and correctness checking; also for state space exploration.
- Knowledge of commutativity properties of operations is essential to practical model checking (it reduces the search space).
- To check for races efficiently, start with a logical specification of commutativity. Convert this commutativity specification into a structural representation. Combine with some specification of happens-before. All of this yields a race detector.
- Example: hashmap. Put, size, and get operations. Specification indicates commutativity using logical formulas on arguments. In a hashmap, insert(i) and insert(j) commute if i != j; for a register, write(i) and write(j) commute if i == j. The latter is harder.
- For more complex objects (e.g., array list from [Deokhwan Kim and Martin Rinard]), commutativity specs can be complicated, so they have devised a scheme to learn the commutativity spec from an abstract (or concrete) implementation.
- The commutativity race detector employs a "micro operations" representation and combines it with a happens-before scheme (vector clocks): it maps conflicts on high-level operations to conflicts on low-level objects, drawing on SIMPLE by [Milind Kulkarni]. It can be tricky to develop a succinct representation. Good representations are important because they enable optimizations of the sort employed by FastTrack for read-write races.
- Conflict checking is $O(n^2)$ in general, but if the conflict predicate is expressed in a particular form, the cost become linear (constant for each new operation). Open question: What is the richest logical fragment that results in linear cost? And which data structures have commutativity specs that lie in that fragment?
- There are various other challenges in both formal specification and and tool construction. For example: Can a read-write race detector precisely detect commutativity races? (Note that space matters.) Also: is there an interesting "consensus-like" hierarchy of concurrency analyzers?

4.29 Application-controlled frequency scaling

Jons-Tobias Wamhoff (TU Dresden, DE)

Joint work of Wamhoff, Jons-Tobias; Diestelhorst, Stephan; Fetzer, Christof; Marlier, Patrick; Felber, Pascal; Dice, Dave
Main reference J.-T. Wamhoff, S. Diestelhorst, C. Fetzer, P. Marlier, P. Felber, D. Dice, "The Turbo Diaries: Application-controlled frequency scaling explained," in Proc. of the 2014 USENIX Annual Technical Conf. (USENIX ATC'14), pp. 193–204, USENIX Association, 2014.
URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/wamhoff

Most multi-/many-core architectures nowadays support dynamic voltage and frequency scaling (DVFS) to adapt their speed to the system's load and save energy. Some recent architectures additionally allow cores to operate at boosted speeds exceeding the nominal base frequency but within their thermal design power. In this talk, we propose a general-purpose library that allows selective control of DVFS from user space to accelerate multi-threaded applications and expose the potential of heterogeneous frequencies. We analyze the performance and energy trade-offs using different DVFS configuration strategies on several benchmarks and real-world workloads. With the focus on performance, we compare the latency of traditional strategies that halt or busy-wait on contended locks and show the power implications of boosting of the lock owner. We propose new strategies that assign heterogeneous and possibly boosted frequencies while all cores remain fully operational. This allows us to leverage performance gains at the application level while all threads continuously execute at different speeds. Our in-depth analysis and experimental evaluation of current hardware provides insightful guidelines for the design of future hardware power management and its operating system interface.

Notes *(collected by members of the audience)*

- Dynamic Voltage and Frequency Scaling (DVFS) leveraging existing X86 multi-cores. Novelty is applying DVFS on the application level.
- Idea of P (Performance) states (pre-defined frequency/voltage pair) and C-states: power states. C0 active; other C states have varying power usage and wake-up time. Trade-off: state transition latency vs. power consumption in that state. Access to states : HLT or MONITOR-MWAIT instructions.
- Investigated AMD Turbo core (modules = frequency domain; AMD: 2 x86 cores + 1 FPU) and Intel Turbo boost (package; both hyperthreads at some frequency). AMD Turbo Core and Intel Turbo boost modes are different. Intel also takes temperature into account. AMD is deterministic by load, can do asymmetric frequencies with manual boost (for one core). Intel reacts to thermal conditions, cores have to be at the same P level. Cores can run at different frequencies on AMD but not Intel. Boosting can be deterministic and thermal; must disable half the cores to give "head-room" to allow it.
- Tested on application Critical Section (CS) benchmark, uses "decorated" MCS lock. Turbo boosted on CS, when it is profitable to do so w.r.t. the overhead of changing V and f. Will (sometimes) trigger DVFS when waiting. Tested both automatic and manual freq scaling; identified the costs of transition. Energy implications of spinning vs. blocking (futex). Goal: run critical section on "fast" CPUs. How big does CS need to be for this to be interesting? Spinning (allows automatic scaling) vs. blocking (OS control). Break even time performance is 1.5M cycles for AMD, about 4M for Intel. Break even for energy is 7M cycle wait time on AMD; much less on Intel. Manual scaling. Overheads in 10s

of 1000s of cycles. Can: spin; owner boost (600k cycles); delegate (dedicated adjustable core); 200k cycles); or migrate (to already boosted core; 400k cycles).
- Developed a turbo library to change P states; simple interface. Ongoing work: boosting STM, async STM up to 50% speedup with 2% energy. Next steps: Haswell-EP supports per-core P-states.
- **Q**: Did you look at Intel SCC processor? It has DVFS domains including the interconnect. **A**: A student is looking into this.

5 Breakout Sessions

5.1 Group Discussion on Heterogeneity

What is a heterogeneous architecture? Numerous sources of heterogeneity are possible, from multiple micro-architectures for a single ISA to integrated CPU/GPU or CellBE-style architectures to fixed function accelerators to FPGA. In addition, heterogeneity exists for communication between near and far components. An old paper on the cyclical nature of display processor design was mentioned.

There was discussion of layers of software that are involved with heterogeneity. For example, in a single-ISA system like ARM big.LITTLE, the OS can migrate threads because all cores share an ISA. In a classic GPU system, the application or perhaps the runtime decides where to run code. The role of each layer (hardware, OS, runtime application) should be considered.

One problem that can arise in heterogeneous systems is poor memory behavior: if accelerators work on data in bulk, then often data must be spilled to DRAM, re-fetched to an accelerator and then re-written to DRAM before it is consumed by another accelerator. Better interfaces between accelerators, or methods to break problems into pieces that fit in the cache, could address this problem. Ideally, the programming model should preserve locality across modules.

Whether accelerators should share the same memory hierarchy as the CPU is also a question; for example, a GPU may trash the CPU cache because of its massive demand for memory bandwidth. There needs to be some control over how cache is shared.

A large concern was who looks out for holistic performance: heterogeneous systems involve many designers who look out for local performance but don't consider the whole system. For example, a GPU may consider that it owns the cache and hurt code on the CPU. When accelerators are integrated on-chip, it may be easier to make tradeoffs because there is tighter integration. In addition, with dark silicon the marginal cost of an accelerator is lower as compared to having to buy an external card.

A large question is when and how to decide where to run code, on which type of core or which accelerator? The ideal, we agreed, was that a programmer writes a single program that is then compiled for multiple architectures. If the placement decision can be made dynamically, then it allows undoing a bad decision. A challenge is that algorithmic changes may be needed to leverage different architectures, such as when moving from a CPU to a GPU, which means the programmer must be involved. The tradeoff here depends on the setting: for a mobile device programmer targeting a heterogeneous set of devices, a single code makes sense. Many programmers today do not want to target GPUs because architecture is moving too fast and they may need to rewrite code in the near future. For Google programmers targeting their own set of machines, then writing multiple versions of the code may be worthwhile.

Ideally, a runtime would dynamically decide what to run where. This is difficult in the face of different ISAs. Locality matters as much as specialization.

Another suggestion was to use libraries: experts can write different versions of a library for different accelerators, and applications can call into the appropriate one. This assumes, though, that the majority of execution time is spent in such libraries today to achieve a good speedup. Furthermore, the overhead of moving data in and out of a library through a procedural interface may be high.

Another concern was the complexity of heterogeneity and lack of predictability. It was noted that Amazon turns off dynamic performance features, such as hyperthreads and turboboost, to provide more predictable performance for customers. It was debated whether cloud vendors would want heterogeneous hardware or prefer homogeneous, as it makes machines more interchangeable and easier to manage. Heterogeneity is coming to the cloud from other places as well, with customized chips from Intel, a mix of DRAM and NVM, different transistor technology, etc.

A final big concern was the complexity of multiple accelerators: it may happen that there are incompatible interfaces, and some pushback may be needed for simpler, more orthogonal interfaces that can be composed easily, even at the loss of some performance.

Selected Contributions to the Discussion

[Torvald Riegel] worried about tool chains. How do we ship code for heterogeneous platforms? How do we integrate code from multiple sources? How do we debug?

[David F. Bacon]: the biggest performance benefits come from the most specialized accelerators. PowerEN suffered from being all wimpy cores.

[Charles E. Leiserson]: Moore's Law is going to end in about 5 years. Read Sutherland's paper [1] on the wheel of reincarnation

[Stephan Diestelhorst]: We tend to write all data to DRAM before starting the next SW module (from different vendor), which then pulls it back into cache. Sharing DRAM with accelerators is clearly good; sharing cache is not so clear.

[Charles E. Leiserson]: Everybody who cares about performance wants to solve the whole problem (in their world, at their level) in a way that writes everybody else out of the equation. Really worried about what this looks like in a world of heterogeneous chips.

[Sven-Bodo Scholz]: Hopeless to expect programmers to cope with heterogeneity.

Q[Michael L. Scott] (strawman): Can we just hide accelerator code behind library interfaces? **A**: [Charles E. Leiserson]: not if they're stateful. **A** [Sven-Bodo Scholz]: and not if state is huge and has to be piped through main memory.

[Milind Kulkarni]: GPUs are a counter-example: we keep data on the GPU across calls. (But this may require compiler help.)

Q: Will heterogeneity permeate the cloud? **A**: [Osman Ünsal]: yes [Charles E. Leiserson]: skeptical. An economic argument: will do it if it's cheaper. [Michael Swift]: Want predictability in billing.

References

1 T. H. Myer and I. E. Sutherland. On the design of display processors. *Commun. ACM*, 11(6):410–414, June 1968. ISSN 0001-0782. DOI: 10.1145/363347.363368.

5.2 Group Discussion on the Future of TM

- How do we make HTM perform well? Issue of cost of start/end transaction on Haswell hardware. Issue of PPC where cost of locking is high, so HTM gains (artificial?) benefit. Unreasonable to expect it to speed up compared with carefully crafted non-blocking algorithms (for example).
- So the space of interest is algorithms for which we do not yet have hand-crafted versions – and goal would be comparable performance.
- One opportunity might be new programming languages – can obtain both simpler code and good performance; on a related point, you can build more sophisticated atomic data structures – such as double-ended queues for work-stealing.
- Need HTM designed to play well with Hybrid schemes.
- Even lock elision is not exactly a "no code changes needed" proposition – consider adding a counter to a critical section: to avoid high rate of abort, it needs to be at the end of the critical section. Observed that a JIT can do this a lot of the time.
- Need debugging and analysis tools that tell developers why aborts occur.
- What about breaking TM down into building blocks with hardware assist? Various program analyses do things quite similar to HTM. Obvious: detection of conflicting accesses. Buffering speculative writes. Ability to pull items back out of a set.
- Non-transactional reads/writes to leak information intentionally.
- Will HTM just fade away? IBM folks think not, but in the Intel space it seems more iffy – consensus is that it needs to be more broadly offered to get more customer usage, but that it appears Haswell TSX will indeed be more broadly available.

5.3 Two Group Discussions on Persistent Memory

Notes from Group 1

[Hans-J. Boehm]: Hardware issues: We must control the order in which things become visible to non-volatile memory. Typically we force data from cache to the memory controller, but not to the non-volatile memory. It seems that the hardware must give you mechanisms to flush the memory controller buffers to memory. Cache line flush instructions (coming on new Intel machines). There are non-temporal store instructions that can be used to write things all the way down to non-volatile memory. There was work at HP that was leveraging "non-temporal stores", which are far from perfect. You may keep multiple versions of the same data in non-volatile memory at the same time. One option is to keep a write-through, non-volatile shadow of volatile DRAM.

[Torvald Riegel]: What impact will there be be on the programming model? 1) Provide a file-system interface – wastes the potential of NVRAM. 2) Libraries and/or 3) Abstraction with loads/stores? We cannot let people to essentially program with persistent loads and stores – cannot use in templates then.

[David F. Bacon]: What are the use cases? Optimize data-base transactional manager? What is the actual application where I will pay the performance overhead for NVRAM and gain?

General agreement that MemCacheD is a nice example (a key-value pair storage). [Torvald Riegel]: Can we write a *portable* nonvolatile memcached?

[Hans-J. Boehm]: There is lots of code in Android to serialize and deserialize data structures. Does NVM mean we can, effectively, just mmap the file that holds the pointer-rich data?

It would be nice to have only one copy of these data structures instead of having to serialize. There will be still need to synchronize with the NVRAM. Can you treat NVRAM simply as a faster way to do serialization? Snapshoting in general as a use case? To make the consistent state persistent. Write a portable C++ for MemCacheD, but need something that the compiler can handle.

System persistence proposed by someone at Microsoft. The idea is to provide enough capacitor/battery power to flush the cache when things are about to die. But may not be able to build a consistent state. The cache-line flushing goes away, the other issues do not go away.

Are transactions the right model? Does it mean that there must be a transactional programming model in place, or is snapshoting sufficient? Can use a lock-based system to obtain the desirable characteristics of a transactional system. We want transactions to protect the integrity of the file system. [Michael L. Scott]: Transactions are atomic methods of concurrent (or stable) objects. The txn system builds bigger abstractions from smaller ones. One seldom (never?) wants pointers from NV to V state.

[David F. Bacon]: But what notation do we give to the programmers to build the NV abstractions?

Many (most? all?) of us have the intuition that persistent pointer-rich structures are "more dangerous" than file-based data. Why, exactly?

Still need partitioning between consistent data and modified data, but we should be able to avoid serialization, so that we do not need to convert into blocks.

How to handle pointers from non-volatile memory to volatile memory? Use a type system?

[Osman Ünsal]: Note that persistent != stable. It doesn't eliminate the need for replication.

[David F. Bacon]: Maybe NVM will be the natural successor for DRAM, for density and cost (and speed?) Maybe persistence will just be a sidelight.

[Michael L. Scott]: Would we then start saying "what can we do with the feature we've been ignoring?" Maybe post-crash forensics of some sort?

Partition memory space so that there is simply a separate partition that is non-volatile.

The window for rollback is much smaller than in the HPC world. And this smaller window is transformative.

HPC-style checkpoint-restart would not work in the case of a buggy program because you do not want to restart in that buggy state.

Is there an opportunity to simply use NVRAM the same way that we use it as a RAM?

Should all storage to persistent state be forced to go through some barrier, such as a system call?

[Hans-J. Boehm]: Lots of people are pursuing APIs similar to mmap (with different implementation under the hood). For example, Facebook has an Mmapped file that persists through the rebooting of a process.

[Michael L. Scott]: What about durable STM? Is there a straightforward path toward adding 'D' to 'ACI'? Are there implications for STM if we add persistence?

[Hans-J. Boehm]: One question is how to do ordering wrt non-transactional accesses – publication and privatization, essentially. If we persist a pointer (transactionally, say), we want to make sure the stuff it points to is already persisted.

Any non-transactional write that is observed by a transaction must be persistent before the transaction is made persistent.

[David F. Bacon]: Maybe we should require *all* persistent writes to be in transactions.

[Torvald Riegel]: But that skips the important optimization of eliding metadata maintenance for update and persistence of the (then) private data.

Note that persisting all previous NV writes of my thread before persisting a txn isn't enough: I have to persist everything that happened before.

[Torvald Riegel]: It's not clear a txnal model is the right level at which to support portability. We may want something lower level.

Should persistence becomes part of the type system?

Focusing on transactional interface, it is hard to pin down semantics for transactions. Would it be better to specify something below that, so that people can then write their transactional abstraction on top of it?

Notes from Group 2

We discussed that while persistence in memory is not new, due to density and power considerations, technologies such as phase change memory/STT-MRAM (reading faster than DRAM/writing slower, 4K reads in 2 μsecs)/memristors will likely replace DRAM. Already, Flash-backed DRAM DIMMs with supercapacitors are available, and in Linux, are treated like a NUMA zone. While typical SSDs are block-based, with kernel-level file system style access, these DIMM replacements will allow fine-grain reads and writes at low latency. The ability to manage durability (persistence) in software is hampered by the high speed.

We discussed what support might be needed, in particular, the ability to "push" data to persistent memory to force durability, and the need for atomicity and ordering. What is needed is:

- Control over when data reaches persistent memory.
- Control over ordering of modifications.
- Evaluating the need for (and ability to avoid) redundant pointers and checksums.

Existing support to flush individual cache lines (e.g., Intel's clflush instruction) could be useful. However, just as clflush interferes with transactions, careful support will be needed to decide on when data is flushed to persistent memory and in what order. To achieve atomicity, maybe create a nonvolatile cache of logs with hardware that later updates the data. We need a combination of volatile and nonvolatile memory in order to control when persistence is attained. How can we simplify the process of synchronously updating data structures and avoid the need for code maintaining these data structures?

We discussed some existing efforts in this direction. [Michael Swift] has provided a bibliography as part of his seminar materials, which is also replicated below). HP Labs did some work on consistent durable data structures. Microsoft has a proposal for whole system persistence via epochs in the cache flush process across all memory controllers in the system and with the ability to explicitly flush the caches when the power goes out. The epoch approach creates a new copy of data that is updated rather than updating in place. Michael Swift's Mnemosyne project moves this to software and writes data at the granularity of an update.

We discussed the need for a "killer" application. Michael Swift summarized that when using NVM for filesystems, in his experiments evaluating a file system benchmark meant as a stress test, only 1% of total accesses were to non-volatile memory (reading/writing data or metadata). One possible application where fine-grain persistence might be useful is high frequency trading, where transactions are small, and all actions are logged for replay by the SEC for possible replay up to 72 hours later. Another is the increasing use of in-memory

databases, where there would be direct reads in processing a query, but still the need to write out logs for persistence. This would reduce commit times from 100 microseconds for flash to 1 microsecond. Could also simplify software.

Other issues to consider and benefits of persistent memory:

- Need for proactive versus reactive support for failure.
- Linux – execute in place – execute operating system from flash.
- Intel's PMFS – directly accesses file data from persistent memory.
- mmap has copy-on-write, persistent memory would need something efficient.
- What about a persistent CAS – in-memory CAS? Or in-cache+extra step for persistence?
- File systems have an important property of consistency and naming; how to retain this?
- Application programming interface to durability needs to change.
- Concurrency control over NVM – managing locks using epoch numbers; epoch mechanism used to create snapshots.
- Persistence as a property of the type system?
- Narrow interface that file systems provide help prevent stray pointer corruption issues; this may be a challenge for fine-grain access to persistent memory.
- Use of a "building block" approach for HTM that can then cover persistence as needed might be beneficial.

References

1. Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. *SIGPLAN Not.*, 47(4):91–104, March 2011. ISSN 0362-1340. DOI: 10.1145/2248487.1950379.
2. Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP'09, pages 133–146, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. DOI: 10.1145/1629575.1629589.
3. Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA'14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-4394-4. DOI: 10.1145/2678373.2665712.
4. Youyou Lu, Jiwu Shu, Long Sun, and O. Mutlu. Loose-ordering consistency for persistent memory. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pages 216–223, Oct 2014. DOI: 10.1109/ICCD.2014.6974684.
5. Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2638-4. DOI: 10.1145/2540708.2540744.
6. Ellis Giles, Kshitij Doshi, and Peter Varman. Bridging the programming gap between persistent and volatile memory using WrAP. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF'13, pages 30:1–30:10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2053-5. DOI: 10.1145/2482767.2482806.
7. Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association. ISBN 978-1-931971-82-9. URL: http://dl.acm.org/citation.cfm?id=1960475.1960480.

6 Panel and Plenary Discussions

6.1 Panel on How to Teach Multi-/Many-core Programming

Panelists: Maurice Herlihy, Charles E. Leiserson, Michael L. Scott, and Nir Shavit.

- Teach concurrency from the beginning.
- Concurrency versus parallelism: avoiding non-determinacy and interaction.
- Carefully distinguish easy cases from the hard ones.
- Can introduce message passage, shared memory, and things like memory models later, in appropriate contexts.
- Need a suitable language for teaching deterministic concurrency in (say) data structures course.
- Java useful because of GC and the concurrency package.
- Abstract algorithms such as Baker's algorithm help give intuition.
- Computer organization has been taught bottom-up and top-down.
- Introduce simple abstractions first.
- We do not have good textbook(s).
- Performance engineering course at MIT teaches parallelism/concurrency in context of larger set of techniques including caching, pipelining, etc.
- Concurrency (interaction) done for reasons not necessarily related to performance; parallelism is for performance.
- Some of this is not about programming – applies to (say) constructing a building.

6.2 Plenary Discussion on VM Design for Concurrency

What primitives should a VM provide? What about threads in a cloud, each from different languages? What sharing of code/metadata would happen between instances of the VM? What do you provide to the language implementer? Channels for communication – can be shared across languages? Great to have hardware help with bookkeeping. Would also be great to be able to capture the logs. May need to capture reads as well. Greater risk if you go too high level as opposed to lower level. How would HTM work in a virtualized environment? Something like Haswell should directly abort. Hazards of a high-level implementation in HW – getting things like page faults, etc., can be problematic. Would it slow things down to use smaller primitives? No intuition that the cost would be higher or lower. Interaction with non-transactional accesses needs to be considered carefully. Park/Unpark has useful properties, and so do futexes – general conditions hard with only futex since it looks at a single location; unpark means you have control over which thread to wake up (can build that with futex as well) Generally want to use these in a style where you re-check the condition (see synchronic<T>). Identities when grabbing a lock or accessing a resource: What is the identity of the agent? Is it a process, thread, whatever? In his talk, [Michael L. Scott] suggests a single hierarchical mechanism – perhaps similar to concurrent nested transactions, or to re-acquisition of the same lock in a subroutine in Java – that is, re-entrant locks depend on a notion of identity. What about accessing "thread-local" storage . . . at different levels of identity – does this get us back into cactus stacks, though maybe needed only for identity and explicitly managed identity-specific storage? Should that be part of the model of the VM? Maybe want something analogous to futexes on identities? Distinction between logical and physical execution agents, e.g., Java thread versus OS thread versus CPU hyperthread –

to what extent would a VM model need to expose this distinction? Could the hierarchy have more to do with what features you have? Along the lines of the concurrency, parallel, and weakly parallel execution agents in the C++ model described in the talk by [Torvald Riegel].

7 Some Results and Open Problems

7.1 Deterministic algorithm for guaranteed forward progress of transactions

Charles E. Leiserson (MIT – Cambridge, US)

The following latest revision of a contention-management algorithm is one of the results of numerous discussions at this Dagstuhl Seminar:

```
SAFE-ACCESS(x)
 1  if lock(x) ∈ L
 2      // do nothing
 3  else
 4      M = {l ∈ L : l > lock(x)}
 5      L = L ∪ {lock(x)}
 6      if M == ∅
 7          ACQUIRE(lock(x)) // blocking
 8      elseif TRY-ACQUIRE(lock(x)) // nonblocking
 9          // do nothing
10      else
11          roll back transaction
12          for l ∈ M
13              RELEASE(l)
14          ACQUIRE(lock(x)) // blocking
15          for all l ∈ M in increasing slot order
16              ACQUIRE(l) // blocking
17          restart transaction // does not return
18  access location x
```

Accessing a memory location x within a transaction with lock set L. The *lock* function maps the space of all locations to a finite ownership array, each slot of which contains an anti-starvation (e.g., queuing) lock. The slots are ordered by an arbitrary linear order, most conveniently, the index in the ownership array. At transaction start, the lock set L is initialized to the empty set: $L = \emptyset$. When the transaction commits, all locks in L are released.

Notes *(collected by members of the audience)*

Deadlock-free because locks on which you wait are acquired in order. On repeated abort, the lock set keeps growing, and that helps with eventual progress. Allows a compilation strategy that figures out locations, acquires in order, and has guaranteed progress. Interesting policy

questions – can do bounded waiting rather than immediately aborting, can wait before restarting, etc. How does this interact with dynamically allocated blocks of memory? Their lock numbers may be different on each try. Probably have a mapping from addresses (say) to a smaller set of lock numbers, which may be some kind of hash – but could (in the SW case) be done in terms of something like Java hash codes, which work even when an object is relocated. The pessimistic style is both its strength and its weakness – but may be able to start optimistic and go pessimistic.

7.2 Thoughts on a Proposal for a Future Dagstuhl Seminar

- Maybe less TM, and more language/tool chain.
- Maybe more people (could have been timing that kept this workshop smaller).
- Heterogeneity should still be part.
- More of the systems-oriented formal methods people.
- Benchmarking and performance evaluation methodology.
- More about abstractions and programming models, and how they might appear in languages.
- In scientific computing domain a new GPGPU + POWER9 machine will be coming available, and that could be relevant.
- The situation around non-volatile storage may be different and that could affect the mix of topics.
- More broadly, the HW picture is evolving, e.g., end of Moore's law.

Participants

- José Nelson Amaral
University of Alberta, CA
- Hagit Attiya
Technion – Haifa, IL
- David F. Bacon
Google – New York, US
- Annette Bieniusa
TU Kaiserslautern, DE
- Hans-J. Boehm
Google – Palo Alto, US
- Daniele Bonetta
Oracle Labs – Linz, AT
- Sebastian Burckhardt
Microsoft Corp. – Redmond, US
- Irina Calciu
Brown University, US
- Dave Dice
Oracle Corp. – Burlington, US
- Stephan Diestelhorst
ARM Ltd. – Cambridge, GB
- Sandhya Dwarkadas
University of Rochester, US
- Pascal Felber
Université de Neuchâtel, CH
- Christof Fetzer
TU Dresden, DE
- Maurice Herlihy
Brown University, US
- Antony Hosking
Purdue University, US
- Milind Kulkarni
Purdue University, US
- Viktor Leis
TU München, DE
- Charles E. Leiserson
MIT – Cambridge, US
- Yossi Lev
Oracle Corp. – Redwood Shores, US
- Alexander Matveev
MIT – Cambridge, US
- Maged M. Michael
IBM TJ Watson Res. Center – Yorktown Heights, US
- J. Eliot B. Moss
University of Massachusetts – Amherst, US
- Erez Petrank
Technion – Haifa, IL
- Michael Philippsen
Univ. Erlangen-Nürnberg, DE
- Torvald Riegel
Red Hat GmbH – Grassbrun, DE
- Paolo Romano
INESC-ID – Lisboa, PT
- Sven-Bodo Scholz
Heriot-Watt University Edinburgh, GB
- Michael L. Scott
University of Rochester, US
- Nir Shavit
MIT – Cambridge, US
- Michael Swift
University of Wisconsin – Madison, US
- Gael Thomas
Télécom & Management SudParis – Evry, FR
- Osman Ünsal
Barcelona Supercomputing Center, ES
- Martin T. Vechev
ETH Zürich, CH
- Jons-Tobias Wamhoff
TU Dresden, DE