Report from Dagstuhl Seminar 17451

New Challenges in Parallelism

Edited by

Annette Bieniusa[1], Hans-J. Boehm[2], Maurice Herlihy[3], and Erez Petrank[4]

1 **TU Kaiserslautern, DE,** bieniusa@cs.uni-kl.de
2 **Google – Palo Alto, US,** hboehm@google.com
3 **Brown University – Providence, US,** mph@cs.brown.edu
4 **Technion – Haifa, IL,** erez@cs.technion.ac.il

Abstract

A continuing goal of current multiprocessor software design is to improve the performance and reliability of parallel algorithms. Parallel programming has traditionally been attacked from widely different angles by different groups of people: Hardware designers designing instruction sets, programming language designers designing languages and library interfaces, and theoreticians developing models of parallel computation. Unsurprisingly, this has not always led to consistent results. Newly developing areas show every sign of leading to similar divergence. This Dagstuhl Seminar will bring together researchers and practitioners from all three areas to discuss and reconcile thoughts on these challenges.

Memory Models and Platforms

Fundamental questions about the semantics of shared memory remain. For example, it becomes increasingly clear that atomic accesses to variables without memory ordering guarantees, or with very weak ordering guarantees, are important in practice. It is surprisingly common to find data structures, such as simple counters, that effectively consist of a single machine word. These continue to be "supported" in languages like Java and C++, but there remains no generally accepted way of defining their semantics, and the specifications in these languages are clearly inadequate. Fundamental questions about memory models and concurrent data structures continue to be unresolved. Many Java concurrent data structures provide weaker than interleaving ("sequentially consistent") semantics that can only be fully understood with a thorough understanding of the memory model. This fact seems to be neither widely appreciated nor discussed. Are the "acquire/release" semantics often used in practice sufficient? Could we afford the overhead of providing the programmer with a simpler model? The more theoretical side of our discipline often uses concepts, such as "safe" and "regular" registers that are quite foreign to the way in which parallel programming languages are actually defined. Are these notions reconcilable?

Non-Volatile Memory and Concurrency

Non-volatile memory (NVM) technologies are expected to support persistence in byte- addressable memory at densities higher than DRAM and at competitive speed. It is expected that NVM will unify the DRAM and SSD into a one-level storage system of persistent main memory with no need for a hard drive, and directly accessible from the programming language. Such a change in the platforms has a significant impact on the design of software and in particular on concurrent algorithms. One implication is that standard functionalities need to be written for a single-tier memory rather then the standard two-tier paradigm. The design of widely available applications, such as database systems, assume that two-tier memory levels are present, and optimizations are based on the fact that these two memory levels have very different behaviors. Concurrency needs to be re-thought in the presence of the new memory structure. Another implication is

New Challenges in Parallelism, *Dagstuhl Reports*, Vol. 7, Issue 11, pp. 1–27
Editors: Annette Bieniusa, Hans-J. Boehm, Maurice Herlihy, and Erez Petrank

Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

that software is now expected to deal with persistence. This has strong connections to thread synchronization issues, but has not been traditionally studied with concurrent algorithms. Addressing these challenges is crucial for building systems on non-volatile memories and we would like to explore potential solutions in the Seminar.

Seminar November 5–10, 2017 – http://www.dagstuhl.de/17451
1998 ACM Subject Classification D.1.3 Concurrent Programming, E.1 Data structures, D.2.4 Software/Program Verification, D.4.2 Storage management
Keywords and phrases concurrency, memory models, non-volatile memory
Digital Object Identifier 10.4230/DagRep.7.11.1
Edited in cooperation with Deepthi Akkoorath

1 Executive Summary

Annette Bieniusa
Hans-J. Boehm
Maurice Herlihy
Erez Petrank

Improving the performance and reliability of parallel and concurrent programs is an ongoing topic in multiprocessor software and hardware research. Despite numerous efforts, the semantics of weak memory models remains subtle and fragile. There has not been established a generally accepted way of defining their semantics, and the specifications of programming languages supporting weak memory models with shared accesses are clearly inadequate. In addition, new advances in hardware are adding further complexity. For example, recently, non-volatile memory (NVM) generated a lot of interest in different communities: Hardware designers coming up with instruction sets and layouts of NVM, system designers integrating NVM into the storage stack, programming language designers proposing library interfaces, and theoreticians developing a new theory of persistence under concurrent access and algorithms adapted for persistency.

This Dagstuhl Seminar on "Future Challenges in Parallelism" touched on different aspects on topics in this broad area of research. In this report, we briefly give a summary of the presentations and discussions that took place during the seminar.

Presentations started with an introductory broad overview talk on the non-volatile memory technology. This survey was followed by shorter talks ranging from hardware techniques for efficiently controlling write-back ordering from caches to theoretical foundations and design of specific data structures.

There was agreement that non-volatile memory is likely to become commercially important in the near future, and that it is tempting to exploit it to provide persistence of user data structures. However, there was little agreement on detailed assumptions and direction. The emphasis of the presentations was on manually designed data structures programmed to a near-hardware-level interface. Some participants expressed concerns that this was too low-level and that the community should instead focus on constructs at the level of durable transactions. Transactional semantics are likely to play an important role: When restarting an application from its persisted state, this state must be consistent in order to prevent data corruption and loss. Much of the work presented in the workshop assumed that we

will have non-volatile memory combined with visibly volatile caches that require explicit flush operations to persist data. But it was pointed out that the problem could be greatly simplified by either providing sufficient battery backup to ensure that the entire cache is flushed on failure, or providing other hardware support.

Some of the participants discussed the definitions of correctness. On the one hand, the standard definition of durable linearizability is a strong requirement that typically brings a large performance overhead. On the other hand, the weaker buffered linearizability does not compose well. Other participants suggested some hardware modifications that could make the life of the programmer easier. For example, a discussion emerged on whether we could pin a cache line to make sure it is not written back to memory. We also tackled the programmability of systems with non-volatile memories. How difficult should it be to program them? Are application programmers expected to employ it directly or only via dedicated data structures provided in libraries? The experience report of porting the application memcached to non-volatile memory raised a lot of interest with the participants. It turned out that the task was rather difficult due to complex interactions between the different modules in the application, in particular between modules that required persistence and modules that did not. The lack of tools was strongly felt, and the obtained performance was not satisfactory. The conclusion was that applications had better be redesigned from scratch to work with non-volatile memory. The general feeling at the end of the seminar was that we are in the beginning of exciting times for research on non-volatile memories and that the discussions must and will continue.

Memory models formed the second major thread of presentations and discussions, with participants expressing widely different viewpoints and technical directions. At one extreme, Madan Musuvathi presented evidence that a simple interleaving-based "sequentially consistent" semantics can be provided at reasonable cost, together with an argument that this is a good direction for future programming languages. At the other extreme, Viktor Vafeiadis argued that a weaker "acquire-release" memory model is easier to reason about, an argument that was backed up by model-checking time measurements. Needless to say, this was followed by lively discussion resulting, we believe in at least a more thorough understanding of different perspectives by everyone involved. There were also several brief presentations and extensive discussion on different approaches for addressing the long-standing C++ and Java (among others) out-of-thin-air problem. Current semantics for these languages allow outcomes that are universally accepted as absurd, but which we do not know how to prohibit in any precise way. It is clear that none of the solutions are quite ready to be adopted, but there are encouraging results along several different paths. There is a consensus that this problem makes formal reasoning about programs nearly impossible and that it is a serious obstruction for tool development. There was less consensus about the extent to which it obstructs day-to-day programming efforts.

In conclusion, the seminar inspired discussions and proposed challenging problems to tackle for the research community. As the discussions showed, designing sound and performant parallel systems require the cooperation of researchers on hardware and software level, with both theoretical and practical analyses and evaluations.

2 Table of Contents

Working groups

3 Overview of Talks

3.1 Coherence, Consistency, and Deja Vu: Memory Hierarchies in the Era of Specialization

Sarita Adve (University of Illinois – Urbana-Champaign, US)

Joint work of Matthew D. Sinclair, Johnathan Alsop, Sarita Adve, and many others
Main reference Matthew D. Sinclair, Johnathan Alsop, Sarita V. Adve: "Efficient GPU synchronization without scopes: saying no to complex consistency models", in Proc. of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015, pp. 647–659, ACM, 2015.
URL http://dx.doi.org/10.1145/2830772.2830821

Memory models for homogeneous multicore systems have traced an arc of complexity that was often precipitated by hardware designs that did not pay enough attention to programmabilty or portability. As we now enter the brave new world of heterogeneous computing and specialization, it is deja vu. We describe how hardware designers are again on a trajectory of exposing too much hardware to software, resulting in complex hierarchies and consistency models. We draw on results from the DeNovo project to show that this trajectory is neither necessary nor effective. With appropriately designed coherence and careful interfaces, we can reap the efficiency benefits of specialization while retaining a simple memory model.

3.2 Global-Local View: Scalable Consistency for Concurrent Data Types

Deepthi Devaki Akkoorath (TU Kaiserslautern, DE)

Joint work of Deepthi Devaki Akkoorath, Annette Bieniusa, Carlos Baquero, José Brandão

Concurrent linearizable access to shared objects can be prohibitively expensive in a high contention workload. Many applications apply ad-hoc techniques to eliminate the need of synchronous atomic updates, which may result in non-linearizable implementations. We propose a *global-local view* model which leverages such patterns for concurrent access to objects in a shared memory system. In this model, each thread maintains different views on the shared object – a thread-local view and a global view. As the thread-local view is not shared, it can be updated without incurring synchronization costs. These local updates become visible to other threads only after the thread-local view is merged with the global view. By executing operations on the local state without synchronization, while only synchronizing with the shared state when needed, applications can achieve better scalability at the expense of linearizability – the default correctness criteria for concurrent objects.

3.3 Remote Memory References at Block Granularity

Hagit Attiya (Technion – Haifa, IL)

Joint work of Hagit Attiya, Gili Yavneh
Main reference To appear in OPODIS 2017.

The cost of accessing shared objects that are stored in remote memory, while neglecting accesses to shared objects that are cached in the local memory, can be evaluated by the number of remote memory references (RMRs) in an execution. We propose a new measure, called block RMRs, counting the number of remote memory references while taking into account the fact that shared objects can be grouped into blocks. On the one hand, this measure reflects the fact that the RMR incurred for bringing a shared object to the local memory might save another RMR for bringing another object placed at the same block. On the other hand, this measure accounts for false sharing: the fact that an RMR may be incurred when accessing an object due to a concurrent access to another object in the same block.

3.4 Analyzing Contention and Backoff in Asynchronous Shared Memory

Naama Ben-David (Carnegie Mellon University – Pittsburgh, US) and Guy E. Blelloch

Joint work of Naama Ben-David, Guy E. Blelloch
Main reference Naama Ben-David, Guy E. Blelloch: "Analyzing Contention and Backoff in Asynchronous Shared Memory", in Proc. of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017, pp. 53–62, ACM, 2017.
URL http://dx.doi.org/10.1145/3087801.3087828

Randomized backoff protocols have long been used to reduce contention on shared resources. They are heavily used in communication channels and radio networks, and have also been shown to greatly improve the performance of shared memory algorithms in real systems. However, while backoff protocols are well understood in many settings, their effect in shared memory has never been theoretically analyzed. This discrepency may be due to the difficulty of modeling asynchrony without eliminating the advantage gained by local delays.

In this talk, I will present a new cost model for contention in shared memory, which allows restricted adversarial asynchrony while capturing the effect of process delays. Using this model, I'll show that we can asymptotically separate the performance of a read-modify-write loop with and without exponential backoff, demonstrating that the model reflects a phenomenon that has evaded rigorous characterization in the past. I'll also introduce a new backoff protocol based on adaptive write-probabilities and show that this protocol outperforms classic exponential backoff under our model.

3.5 Concurrency as First-class Entity

Annette Bieniusa (TU Kaiserslautern, DE)

Joint work of Mathias Weber, Annette Bieniusa, Arnd Poetzsch-Heffter
Main reference Mathias Weber, Annette Bieniusa, Arnd Poetzsch-Heffter: "EPTL - A Temporal Logic for Weakly Consistent Systems (Short Paper)", in Proc. of the Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings, Lecture Notes in Computer Science, Vol. 10321, pp. 236–242, Springer, 2017.
URL https://doi.org/10.1007/978-3-319-60225-7_17

Concurrency is a natural phenomenon. Information is essentially local to a process and becomes only visible once processes interact and communicate. Yet, programming models are typically choosing linearizability as standard semantics.

In my talk I am presenting a novel temporal logic, event-based parallel temporal logic (EPTL), that allows to reason about weakly-consistent systems. In contrast to other temporal logics like LTL or CTL, EPTL allows to model semantics of components that are truly concurrent while abstracting from implementation and communication details.

3.6 What consistency guarantees should concurrent data structure libraries provide?

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

Main reference Hans-J. Boehm, P0387R0: Memory Model Issues for Concurrent Data Structures (WG21 standards committee paper)
URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0387r0.html

The C++ standards committee is is attempting to add concurrent data structures, for example concurrent queues, to the standard library. This has raised interesting questions about desired correctness properties for such libraries. The traditional linearizability criterion is not entirely consistent with the underlying C++ memory model, and does not sufficiently address interactions with visibility of simple assignments. This is complicated by the fact that there is little agreement on the correct visibility guarantees, and that they seem to vary between data structures. We briefly addressed some of the issues, tradeoffs, and challenges.

3.7 Remote memory in the age of fast networks

Irina Calciu (VMware – Palo Alto, US)

Joint work of Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, Michael Wei
Main reference Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, Michael Wei: "Remote memory in the age of fast networks", in Proc. of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017, pp. 121–127, ACM, 2017.
URL http://dx.doi.org/10.1145/3127479.3131612

As the latency of the network approaches that of memory, it becomes increasingly attractive for applications to use remote memory–random-access memory at another computer that is accessed using the virtual memory subsystem. This is an old idea whose time has come, in the age of fast networks. To work effectively, remote memory must address many technical challenges. We hope to provide a broad research agenda around this topic, by proposing more problems than solutions.

3.8 NVM ReConstruction: Object-Oriented Recovery for Non-Volatile Memory

Nachshon Cohen (EPFL – Lausanne, CH)

Joint work of Nachshon Cohen, Virendra Marathe, James Larus

New non-volatile memory (NVM) technologies allow direct, durable storage of data in an application's heap. This type of random-access memory can simplify the construction of reliable applications that do not lose data at a system shutdown or power failure. Existing NVM programming frameworks for native languages are applicable to non-object oriented languages such as C, and do not gracefully support richer abstractions and constructs available in an object-oriented language like C++. Support for even mundane abstractions, such as pointers, leads to highly error prone programming practices. This paper presents a new persistent memory programming model designed to naturally align with the object-oriented programming style, while addressing programming pitfalls of prior approaches. At the heart of our approach is the notion of *object reconstruction*, the ability to transparently reconstruct a persistent object's state at runtime after a restart event. Object reconstruction enables several desirable features that significantly simplify the programmer's task of writing correct programs that leverage byte-addressability of persistent memory. It (i) enables support for key object-oriented features such as type inheritance and virtual functions in persistent types, (ii) accommodates co-location of nonpersistent fields within persistent type instances, (iii) allows representation of persistent pointers as virtual addresses, and (iv) enables type specific reconstruction of the entire state of a persistent object after a restart. Our prototype implementation as a C++ library demonstrates the versatility of object reconstruction at virtually zero performance overhead.

3.9 Waiting Policies

Dave Dice (Oracle Corp. – Burlington, US)

Main reference Dave Dice: "Malthusian Locks", CoRR, Vol. abs/1511.06035, 2015.
URL http://arxiv.org/abs/1511.06035

The "best practice" for waiting – threads waiting for another thread to change a location in shared memory – has shifted with modern architectures. We briefly survey the motivating architectural factors and suggest new practices.

3.10 Performance Isolation on Modern Multi-Socket Systems

Sandhya Dwarkadas (University of Rochester, US)

Joint work of Sharanyan Srikanthan and Kai Shen
URL http://www.cs.rochester.edu/u/sandhya

Recognizing that parallel applications are rarely executed in isolation today, I will discuss some practical challenges in making best use of available hardware and our approach to addressing these challenges. I describe two independent control mechanisms: a sharing- and resource-aware mapper (SAM) to effect task placement with the goal of localizing shared data communication and minimizing resource contention based on the offered load; and an application parallelism manager (MAP) that controls the offered load with the goal of improving system parallel efficiency. Our results emphasize the need for low-overhead monitoring of application behavior under changing environmental condiitons in order to adapt to environment and application behavior changes.

References

1 Coherence Stalls or Latency Tolerance: Informed CPU Scheduling for Socket and Core Sharing, Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen, Usenix Annual Technical Conference (Usenix ATC), Denver, CO, June 2016.
2 Data Sharing or Resource Contention: Toward Performance Transparency on Multicore Systems, Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen, Usenix Annual Technical Conference (Usenix ATC), Santa Clara, CA, July 2015.

3.11 Efficient Distributed Data Structures for Future Many-core Architectures

Panagiota Fatourou (University of Crete – Heraklion, GR), Nikolaos D. Kallimanis, Eleni Kanellou, Odysseas Makridakis, and Christi Symeonidou

Main reference Panagiota Fatourou, Nikolaos D. Kallimanis, Eleni Kanellou, Odysseas Makridakis, Christi Symeonidou: "Efficient Distributed Data Structures for Future Many-Core Architectures", in Proc. of the 22nd IEEE International Conference on Parallel and Distributed Systems, ICPADS 2016, Wuhan, China, December 13-16, 2016, pp. 835–842, IEEE Computer Society, 2016.
URL http://dx.doi.org/10.1109/ICPADS.2016.0113

We study general techniques for implementing distributed data structures on top of future many-core architectures with non cache-coherent or partially cache-coherent memory. With the goal of contributing towards what might become, in the future, the concurrency utilities package in Java collections for such architectures, we end up with a comprehensive collection of data structures by considering different variants of these techniques. To achieve scalability, we study a generic scheme which makes all our implementations hierarchical. We also describe a collection of techniques for further improving scalability in most implementations. We have performed experiments which illustrate nice scalability characteristics for some of the proposed techniques and reveal the performance and scalability power of the hierarchical approach. We distill the experimental observations into a metric that expresses the scalability potential of such implementations. We finally present experiments to study energy consumption aspects of the proposed techniques by using an energy model recently proposed for such architectures.

3.12 Persistent Lock-Free Data Structures for Non-Volatile Memory

Michal Friedman (Technion – Haifa, IL)

Joint work of Michal Friedman, Erez Petrank, Maurice Herlihy, Virendra Marathe, Nachshon Cohen
Main reference Michal Friedman, Maurice Herlihy, Virendra J. Marathe, Erez Petrank: "Brief Announcement: A Persistent Lock-Free Queue for Non-Volatile Memory", in Proc. of the 31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria, LIPIcs, Vol. 91, pp. 50:1–50:4, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
URL http://dx.doi.org/10.4230/LIPIcs.DISC.2017.50

Non-volatile memory is expected to coexist with (or even displace) volatile DRAM for main memory in upcoming ar- chitectures. This has led to increasing interest in the problem of designing and specifying durable data structures that can recover from system crashes. Data structures may be designed to satisfy stricter or weaker durability guarantees to provide a balance between the strength of the provided guarantees and performance overhead. The talk proposes three novel implementations of a concurrent lock-free data structures. These implementations illustrate algorithmic challenges in building persistent lock-free data structures with different levels of durability guarantees. In presenting these challenges, the pro- posed algorithmic designs, and the different durability guaran- tees, we hope to shed light on ways to build a wide variety of durable data structures.

3.13 Everything Better Than Everything Else

Tim Harris (Oracle Labs – Cambridge, GB)

Evaluating shared-memory data structures is difficult: there are lots of possible metrics, lots of possible ways to run experiments, and lots of ways in which low-level details of the hardware can have unexpectedly large impacts on performance. It is hard to untangle algorithmic improvements from coding prowess. I am going to talk about some of the ways in which I have been bitten by these problems in the past, and some of the techniques I use to try to organize my own experimental work.

3.14 Recoverable Mutual Exclusion in Sub-logarithmic Time

Danny Hendler (Ben Gurion University – Beer Sheva, IL) and Wojceich Golab

Joint work of Danny Hendler, Wojciech Golab
Main reference Wojciech M. Golab, Danny Hendler: "Recoverable Mutual Exclusion in Sub-logarithmic Time", in Proc. of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017, pp. 211–220, ACM, 2017.
URL http://dx.doi.org/10.1145/3087801.3087819

Recent developments in non-volatile main memory (NVRAM) media foreshadow the eventual convergence of primary and secondary storage into a single layer in the memory hierarchy that combines the performance benefits of conventional main memory with the durability of secondary storage. Traditional log-based recovery techniques can be applied correctly in such systems but fail to take full advantage of the parallelism enabled by allowing processing cores to access recovery data directly using memory operations rather than slow block transfers from secondary storage. As a result, harnessing the performance benefts of NVRAM-based platforms requires a careful rethinking of recovery mechanisms. Recoverable mutual exclusion (RME) is a variation on the classic mutual exclusion (ME) problem that allows processes to crash and recover. Prior work on the RME problem has established an upper bound of O(log N) remote memory references (RMRs) in an asynchronous shared memory model with N processes that communicate using atomic read and write operations, prompting the question whether sub-logarithmic RMR complexity is attainable using commonly supported read-modify-write primitives. We answer this question positively by presenting an RME algorithm that incurs O(log N/ log log N) RMRs in the cache-coherent model and uses standard read, write, Fetch-And-Store, and Compare-And-Swap instructions. The algorithm uses as a building block a new recoverable extension of Mellor-Crummey and Scott's queue lock that is interesting in its own right. We also present an O(1) RMRs algorithm that relies on double-word Compare-And-Swap and a double-word variation of Fetch-And-Store.

3.15 Hybrid STM/HTM for Java

Antony Hosking (Australian National University – Canberra, AU)

Joint work of Keith Chapman, Antony Hosking, Eliot Moss
Main reference Keith Chapman, Antony L. Hosking, J. Eliot B. Moss: "Hybrid STM/HTM for nested transactions on OpenJDK", in Proc. of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016, pp. 660–676, ACM, 2016.
URL http://dx.doi.org/10.1145/2983990.2984029

Our work on hybrid STM/HTM for Java allows concurrently executing transactions to use low-cost HTM when it works, but revert to STM when it doesn't, even as other transactions are running STM/HTM. Our implementation is for an extension of Java having syntax for both open and closed nested transactions, and boosting, running on the OpenJDK. We demonstrate that HTM offers significant acceleration of both closed and open nested transactions, while yielding parallel scaling up to the limits of the hardware, whereupon scaling in software continues but with the usual penalty to throughput imposed by software mechanisms.

A useful takeaway from our work is the extent that open nesting enables more profitable use of HTM.

3.16 Concurrent data structures for scalable real-time analytics

Idit Keidar

Joint work of Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, H. Porat, A. Spiegelman, Moshe Sulamy
Main reference Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, Moshe Sulamy: "KiWi: A Key-Value Map for Scalable Real-Time Analytics", in Proc. of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017, pp. 357–369, ACM, 2017.
URL http://dl.acm.org/citation.cfm?id=3018761

Modern big data processing platforms employ huge in-memory key-value (KV) maps. Their applications simultaneously drive high-rate data ingestion and large-scale analytics. These two scenarios expect KV-map implementations that scale well with both real-time updates and large atomic scans triggered by range queries.

I will discuss research efforts at Yahoo Research addressing this challenge in the context of Druid, a high-performance, column-oriented, distributed data store. I will first discuss recently published work on KiWi – A Key-Value Map for Scalable Real-Time Analytics [1]. I will then discuss two on going projects: Oak – Off-Heap Allocated Keys, and Concurrent Data Sketches.

References

1 Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. In Sarkar and Rauchwerger [2], pages 357–369.

2 Vivek Sarkar and Lawrence Rauchwerger, editors. *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*. ACM, 2017.

3.17 Generic Concurrency Restriction

Alex Kogan (Oracle Labs – Burlington, US)

Joint work of Dave Dice, Alex Kogan

Contended locks often degrade the performance of a multithreaded application, leading to a so-called scalability collapse problem. This problem arises when a growing number of threads circulating through a saturated lock causes the overall application performance to fade or even drop abruptly. In this talk, I will introduce GCR (generic concurrency restriction), a mechanism that aims to avoid the scalability collapse. GCR, designed as a generic, lock-agnostic wrapper, intercepts lock acquisition calls, and decides when threads would be allowed to proceed with the acquisition of the underlying lock. Furthermore, I will describe GCR-NUMA, an adaptation of GCR for non-uniform memory access (NUMA) settings.

3.18 Flat Combining and Transactional Lock Elision – combining pessimistic and optimistic mechanisms together

Yossi Lev (Oracle Labs – Burlington, US)

Joint work of Alex Kogan, Yossi Lev
Main reference Alex Kogan, Yossi Lev: "Transactional Lock Elision Meets Combining", in Proc. of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017, pp. 231–240, ACM, 2017.
URL http://dx.doi.org/10.1145/3087801.3087838

Flat combining (FC) is a technique that can significantly improve the performance of operations that conflict with each other when accessing a shared data structure – e.g., they all trying to modify the same field of the data structure. Transactional Lock Elision (TLE), on the other hand, significantly improve the performance when the operations applied to the data structure rarely conflict – that is, in the common case, they access different parts of the data structures; this is done by using an optimistic approach where hardware transactions execute the operations in parallel, and retry when they conflict with one another. Both techniques provides the desired property of allowing the programmer to write simple, sequential code that does not need to handle interference by other operations running in parallel – as if the operations are ran under a lock protecting all accesses to the data structure. In this presentation I show how we can combine the two techniques when dealing with data structures where some of their operations are likely to conflicts with each other, while others do not, and can there run in parallel using transactional lock elision. The new technique keeps the simplicity of programming without needing to reason about interference with other operations running in parallel, and significantly improves the performance comparing to use either of these techniques by its own.

3.19 Verifying Concurrent GC Running in Weakly Ordered Memories

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

We are in the process of developing a suite of state-of-the-art concurrent garbage collectors for a language-independent virtual machine, intended to simply the implementation of managed languages. This VM, called Mu (for its efforts to be small, i.e., a "micro" VM), has an intermediate language similar to LLVM (close to hardware, but with unbounded local "registers" and SSA-form, and supporting C/C++-style memory-access ordering specifications) but tailored and extended for managed language support in a concurrent environment. Mu is targeting a range of modern processors, including x86 and ARM. Thus the GC must deal with hardware that reorders memory accesses.

Challenges in mechanical verification of this suite of GCs include the sheer size and subtlety of the necessary code, and he desire to reuse, as much as possible, proofs when dealing with code variations of largely local impact. What kinds of specifications and logics will aid in this effort? What proof strategies will make the size of the tasks manageable?

3.20 How much does sequential consistency cost anyway

Madan Musuvathi (Microsoft Research – Redmond, US)

Joint work of Madan Musuvathi, Lun Liu, Dan Marino, Todd Millstein, Satish Narayanaswamy, Ryan Newton, Ryan Scott, Abhay Singh, Michael Vollmer
Main reference Lun Liu, Todd D. Millstein, Madanlal Musuvathi: "A volatile-by-default JVM for server applications", PACMPL, Vol. 1(OOPSLA), pp. 49:1–49:25, 2017.
URL http://dx.doi.org/10.1145/3133873

Research on weak memory-models assume that sequential consistency (SC) is expensive for modern languages on modern hardware. Over the past few years, we have attempted to empirically answer the question: how much does SC cost anyway? While some of the results are what one would expect, others are surprising, at least to us. In this talk I will describe these results and what they might imply:

a) turning off C/C++ compiler optimizations that violate SC results in little performance overhead. This implies that complexities in memory-model design that solely arise due to compiler optimizations can be avoided

b) when done carefully, SC for Java results in an overhead of 12-28% on Intel architectures, and recent experiments suggest similar overheads on ARM architectures as well. Significant part of this overhead comes from inserting hardware fences in a few core libraries. This suggests that Java should allow programmers to escape into relaxed semantics only for carefully chosen code segments, just as it allows programmers to selectively escape into type-unsafe segments.

c) SC has no perceptible overheads for Haskell and possibly for other functional programming languages. This might imply that the strict type isolation that these languages provide between pure and imperative parts of a program could be incorporated into mainstream languages to simplify memory models.

3.21 Efficient Architectural Support for Persistent Memory

Vijay Nagarajan (University of Edinburgh, GB)

Joint work of Arpit Joshi, Vijay Nagarajan, Stratis Viglas, Marcelo Cintra
Main reference Arpit Joshi, Vijay Nagarajan, Stratis Viglas, Marcelo Cintra: "ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging", in Proc. of the 2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017, pp. 361–372, IEEE Computer Society, 2017.
URL http://dx.doi.org/10.1109/HPCA.2017.50
Main reference Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, Stratis Viglas: "Efficient persist barriers for multicores", in Proc. of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015, pp. 660–671, ACM, 2015.
URL http://dx.doi.org/10.1145/2830772.2830805

Emerging persistent memory technologies enable fast, fine-grained durability compared to slow block-based devices. Programming with persistent memory, however, requires primitives that provide guarantees about what has been made durable. Two primitives that programmers understand well are: ordering (persist barrier) and atomicity (atomic durable transactions). A persist barrier guarantees that stores before the barrier, update persistent memory before stores that follow it. An atomic durable transaction guarantees that stores within the transaction update persistent memory atomically. In this talk, we will show how these two primitives can be efficiently implemented using architectural support.

3.22 Concurrent Data structures for Non-Volatile Memory

Erez Petrank (Technion – Haifa, IL)

Joint work of Erez Petrank, Michal Friedman, Nachshon Cohen, Maurice Herlihy, Virendra Marathe

Non-volatile memory is expected to coexist with (or even displace) volatile DRAM for main memory in upcoming architectures. This has led to increasing interest in the problem of designing and specifying durable data structures that can recover from system crashes. Definitions for durable linearizability have been recently proposed by Israelevitz et al. They have also proposed an automatic transformation for making lock-free data structures resilient to crashed on non-volatile memories. However, the automatic construction is typically inefficient. In this lecture we report an effort to design data structures that are appropriate for this setting, and also discuss some additional definitions.

3.23 Efficient Inspected Critical Sections in data-parallel GPU codes

Michael Philippsen (Universität Erlangen-Nürnberg, DE) and Thorsten Blass

Main reference Thorsten Blass, Michael Philippsen, Ronald Veldema: "Efficient Inspected Critical Sections in data-parallel GPU codes", in Proc. of the 30th Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC 2017), to be published as LNCS volume, Springer, 2018.

Optimistic concurrency control and STMs rely on the assumption of sparse conflicts. For data-parallel GPU codes with many or with dynamic data dependences, a pessimistic and lock-based approach may be faster, if only GPUs would offer hardware support for GPU-wide

fine-grained synchronization. Instead, current GPUs inflict dead- and livelocks on attempts to implement such synchronization in software.

The paper demonstrates how to build GPU-wide non-hanging critical sections that are as easy to use as STMs but also get close to the performance of traditional fine-grained locks. Instead of sequentializing all threads that enter a critical section, the novel programmer-guided Inspected Critical Sections (ICS) keep the degree of parallelism up. As in optimistic approaches threads that are known not to interfere, may execute the body of the inspected critical section concurrently.

3.24 The old challenge: How to support users?

Mirko Rahn (Fraunhofer ITWM – Kaiserslautern, DE)

In High Performance Computing many application developers are experts in their current domain rather than in computer science. To deal with parallelism adds a layer of complexity that is not easy to manage for them. We show some common misconceptions that occur in real life programs. Some of the bugs are easy to spot for well trained readers. So where are the tools that support the users?

3.25 Durable Linearizability

Michael L. Scott (University of Rochester, US)

Joint work of Joseph Izraelevitz, Hammurabi Mendes, Michael L. Scott
Main reference Joseph Izraelevitz, Hammurabi Mendes, Michael L. Scott: "Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model", in Proc. of the Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings, Lecture Notes in Computer Science, Vol. 9888, pp. 313–327, Springer, 2016.
URL https://doi.org/10.1007/978-3-662-53426-7_23

The prospect of ubiquitous nonvolatile main memory suggests the possibility of maintaining long-lived data unmediated by the file system, but only if data is carefully managed to ensure consistency in the wake of a crash. In keeping with "real world" systems, we introduce the notion of *durable linearizability* to govern the safety of concurrent objects when all transient state (of all threads) is lost on a crash; we also introduce a buffered variant in which the recoverable state is consistent but not necessarily up to date. At the implementation level, we present *explicit epoch persistency*, a formal model that builds upon and generalizes prior work, together with an automated transform to convert any linearizable, nonblocking but data-race–free concurrent object into one that is (buffered) durably linearizable. We also present a design pattern, analogous to linearization points, for the construction of other, more optimized objects. On top of this formal foundation, we present a series of software mechanisms–JUSTDO logging, iDO logging, and periodic persistence–for fast, composable persistence.

3.26 Just-Right Consistency: As available as possible, consistent when necessary

Marc Shapiro (University Pierre & Marie Curie – Paris, FR), Annette Bieniusa, Christoper Meiklejohn, Nuno Preguiça, and Valter Balegas

In a distributed data store, the CAP theorem forces a choice between strong consistency (CP) and availability and responsiveness (AP) when the network can partition. To address this issue, we take an application-driven approach, Just-Right Consistency (JRC). JRC defines a consistency model that is sufficient to maintain the application's specific invariants, and otherwise remaining as available as possible.

JRC leverages knowledge of the application. Two invariant-maintaining programming patterns, ordered updates and atomic grouping, are compatible with asynchronous updates, orthogonally to CAP. In contrast, checking a data precondition on replicated state is CAP-sensitive. However, if two updates do not negate each other's precondition, they may safely execute concurrently. Updates must synchronise only if one negates the precondition of the other.

The JRC approach is supported by the CRDT data model that ensures that concurrent updates converge; by Antidote, a cloud-scale CRDT data store that guarantees transactional causal consistency; and by developer tools (static analysers and domain-specific languages) that help guarantee invariants.

3.27 Causal atomicity

Simon Doherty, John Derrick, Brijesh Dongol, Heike Wehrheim

Correctness conditions for concurrent objects (like linearizability or opacity) require "seemingly atomic", though concurrent, access to shared memory. So far, these correctness conditions build on an interleaving model of concurrency: concurrent executions are represented as totally ordered sequences of invocations and returns of operations (so called histories). Behind this concept of histories is the assumption of a notion of global time and the total observability of orderings of operations. This assumption, however, fails to hold for weak memory models in which there are only partial orders on operations, as for instance generated by a happens-before relation.

We propose a generalization of concurrent correctness conditions to weak memory called causal atomicity. It is based on Lamport's execution structures which are sets of events together with two relations: a partially ordered precedence relation together with a relation describing communication among events (like reads-from relations). Causal atomicity compares execution structures to sequential specifications. Alike linearizability and opacity, we have shown causal atomicity to be compositional in that the composition of causally atomic objects yields causally atomic structures.

3.28 Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems

Matthew Sinclair (AMD Research – Bellevue, US)

Joint work of Matthew D. Sinclair, Johnathan Alsop, Sarita V. Adve
Main reference Matthew D. Sinclair, Johnathan Alsop, Sarita V. Adve: "Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems", in Proc. of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017, pp. 161–174, ACM, 2017.
URL http://dx.doi.org/10.1145/3079856.3080206

An unambiguous and easy-to-understand memory consistency model is crucial for ensuring correct synchronization and guiding future design of heterogeneous systems. In a widely adopted approach, the memory model guarantees sequential consistency (SC) as long as programmers obey certain rules. The popular data-race-free-0 (DRF0) model exemplifies this SC-centric approach by requiring programmers to avoid data races. Recent industry models, however, have extended such SC-centric models to incorporate relaxed atomics. These extensions can improve performance, but are difficult to specify formally and use correctly. This work addresses the impact of relaxed atomics on consistency models for heterogeneous systems in two ways. First, we introduce a new model, Data-Race-Free-Relaxed (DRFrlx), that extends DRF0 to provide SC-centric semantics for the common use cases of relaxed atomics. Second, we evaluate the performance of relaxed atomics in CPU-GPU systems for these use cases. We find mixed results – for most cases, relaxed atomics provide only a small benefit in execution time, but for some cases, they help significantly (e.g., up to 51% for DRFrlx over DRF0).

3.29 Memory Instrumentation As A First-Class Language Feature

Michael F. Spear (Lehigh University – Bethlehem, US)

Joint work of Michael F. Spear, Xiaochen Guo, Aviral Shrivastava, Gang Tan

Across a wide range of domains, researchers require the ability to quickly instrument specific loads and stores in a program, in order to achieve novel memory behaviors. In this talk, I will introduce the first fruit of the Abstract Instrumented Memory Interface (AIMI) project: an LLVM plugin that allows researchers to perform region-based memory instrumentation of arbitrary C and C++ programs. I will first discuss motivation, then implementation, and finally discuss performance and design tradeoffs, with a focus on the AIMI submodule for Transactional Memory.

3.30 Sequential consistency considered harmful

Viktor Vafeiadis (MPI-SWS – Kaiserslautern, DE)

Sequential consistency (SC) is typically presented as the ideal semantics for shared-memory concurrency, because it is has a simple and understandable definition. This definitional simplicity, however, does not necessarily extend when SC accesses are incorporated in a weak memory model; see, e.g., [1]. In this talk, I highlighted some of the pitfalls of SC, and why it might not be such an ideal model after all. I further discussed *release/acquire* consistency (RA) [2] as an alternative model that has some tangible benefits over SC and may thus be a better model for shared-memory concurrency.

First, I showed two automated verification problems that are easier to solve for RA than for SC. Specifically, checking whether a given execution of a program annotated with reads-from edges is consistent is an NP-complete problem for SC, but is decidable in polynomial time for RA. As a result, bounded model checking for RA can be made to run significantly faster than the state of the art SC model checking tools; see, e.g., [3].

Second, regarding manual software verification, I argued that RA permits the two most useful forms of reasoning–namely *local* reasoning and *causal* reasoning–which are nicely encompassed in concurrent separation logics, such as RSL [4] and GPS [5, 6]. On the other hand, it forbids *global* reasoning [7]; i.e. reasoning in terms of which action of a thread was executed first. Global reasoning, which is sound according to SC, is a poor form of reasoning about the correctness of programs: it is complicated and error-prone; and as such is rarely used to prove the correctness of concurrent programs. By forbidding this kind of reasoning, therefore, RA may actually be a better programming model.

Finally, I briefly argued that *multicopy atomicity*–a crucial aspect of SC–prevents scalability, because it enforces a global order on every two totally independent writes by independent processors. Yet, multicopy atomicity does not seem relevant for reasoning about the correctness of concurrent programs and could thus be dropped without a perceivable difference in semantics.

References

1 Repairing sequential consistency in C/C++11. Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, Derek Dreyer. In PLDI 2017, pp. 618–632. ACM 2017.
2 Ori Lahav, Nick Giannarakis, Viktor Vafeiadis. Taming release-acquire consistency. In POPL 2016, pp. 649–662. ACM, 2016.
3 Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. Proc. ACM Program. Lang. 2, POPL, Article 17. ACM, 2018.
4 Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. In OOPSLA 2013, pp. 867–884. ACM, 2013.
5 Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In OOPSLA 2014, pp. 691–707. ACM, 2014.
6 Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In ECOOP 2017. LIPIcs, vol. 74, pp. 17:1–17:29. Schloss Dagstuhl, 2017.
7 Ori Lahav and Viktor Vafeiadis. Owicki-Gries Reasoning for Weak Memory Models. In ICALP 2015, Part II. LNCS, vol. 9135, pp. 311–323. Springer, 2015.

3.31 Applying lock-free data structures to fabric-attached memory

Haris Volos (HP Labs – Palo Alto, US)

Next-generation rack-scale architectures will enable a fabric-attached non-volatile memory pool accessible by all compute resources. Fabric-attached memory will make interesting new programming styles possible. In this talk, I will report on our experience with one such style obtained by applying lock-free data-structure techniques to fabric-attached memory.

3.32 Isoefficiency in Practice: Configuring and Understanding the Performance of Task-based Applications

Felix Wolf (TU Darmstadt, DE)

Joint work of Sergei Shudler, Alexandru Calotoiu, Torsten Hoefler, Felix Wolf
Main reference Sergei Shudler, Alexandru Calotoiu, Torsten Hoefler, Felix Wolf: "Isoefficiency in Practice: Configuring and Understanding the Performance of Task-based Applications", in Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Austin, TX, USA, pp. 131–143, ACM, February 2017.
URL http://dx.doi.org/10.1145/3018743.3018770

Task-based programming offers an elegant way to express units of computation and the dependencies among them, making it easier to distribute the computational load evenly across multiple cores. However, this separation of problem decomposition and parallelism requires a sufficiently large input problem to achieve satisfactory efficiency on a given number of cores. Unfortunately, finding a good match between input size and core count usually requires significant experimentation, which is expensive and sometimes even impractical. In this paper, we propose an automated empirical method for finding the isoefficiency function of a task-based program, binding efficiency, core count, and the input size in one analytical expression. This allows the latter two to be adjusted according to given (realistic) efficiency objectives. Moreover, we not only find (i) the actual isoefficiency function but also (ii) the function one would yield if the program execution was free of resource contention and (iii) an upper bound that could only be reached if the program was able to maintain its average parallelism throughout its execution. The difference between the three helps to explain low efficiency, and in particular, it helps to differentiate between resource contention and structural conflicts related to task dependencies or scheduling. The insights gained can be used to co-design programs and shared system resources.

3.33 Effect Summaries for Thread-Modular Analysis

Sebastian Wolff (TU Braunschweig, DE), Lukáš Holík, Roland Meyer, and Tomáš Vojnar

Main reference Lukás Holík, Roland Meyer, Tomás Vojnar, Sebastian Wolff: "Effect Summaries for Thread-Modular Analysis - Sound Analysis Despite an Unsound Heuristic", in Proc. of the Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings, Lecture Notes in Computer Science, Vol. 10422, pp. 169–191, Springer, 2017.
URL https://doi.org/10.1007/978-3-319-66706-5_9

Thread-modular verification is the state of the art approach for verifying lock-free data structures. However, existing approaches have problems with scalability when applied in environments with manual memory management (i.e. no garbage collection).

To overcome this limitation, we identified a common programming idiom in lock-free data structures: so-called copy-and-check blocks. We show how to exploit such patterns for thread-modular analyses and report on our findings.

4 Working groups

4.1 Distributed Concurrency

Michal Friedman (Technion – Haifa, IL) and Virendra Marathe (Oracle Corp. – Burlington, US)

Summary:

The discussion, though meandering through several different topics, seemed to gravitate toward the similarities between shared memory and distributed systems. The final takeaway was that with ongoing and future technology changes, distributed systems are looking much more like shared memory systems, and vice versa. This unlocks the potential to apply wisdom learned in one setting to the other.

Details:

Maurice Herlihy's observations of similarities between the environments and challenges faced today by the world of Blockchains has strong resonance with the world of concurrency. Many of the solutions applied to the latter seem applicable to the former.

What hardware offers you and how it can be leveraged? For instance, for many-core NUMA systems, Barrelfish presents a distributed view of a shared memory system. Maybe we can think of NUMA systems as distributed systems (which is already happening).

Can lessons learned from distributed system settings be applied in shared memory setting? Today's multi-core systems are essentially distributed, with larger systems and different coherence domains (with accelerators showing up more and more in our systems), and explicit programmer intervention between these different domains.

Should we make a distinction between distributed systems and shared memory systems? There are several points in the spectrum. Even when you are presenting a simple shared memory model, discussing the implementation complexities that go into making that happen would be an interesting topic. Trade offs of programming complexity: make locality simpler at the expense of going through hoops for global access, or vice versa. In another words,

better locality vs. complicated communication. Also, people have the knowledge to transfer from multicore to distributed systems. Can they do the opposite? yes!

An important difference between distributed and shared memory systems is their failure models – fail-stop models for shared memory systems, whereas the failure models of distributed systems emphasize fault tolerance, including hardware failure tolerance. However, fault isolation and security in the new containerized cloud computing world is another form of distributed computing specific issues that are applicable to shared memory systems.

Furthermore, modern hardware trends toward fast and direct remote access (RDMA) is changing the interface to distributed systems as well, where they are now looking increasingly like shared memory NUMA systems. As the cost of communication drops with these technologies, the incentive to make interfaces look similar to a single shared memory system increases.

4.2 Relaxed Atomicity

Madan Musuvathi (Microsoft Research – Redmond, US)

During the Dagstuhl we had two great discussions on weak memory models and the challenges in both providing an easy to understand interface to programmers while allowing the compiler and hardware enough freedom to implement useful optimizations. While performance of a memory model can be easily measured, a measure of its naturalness or ease of understanding to programmers is much harder to define. One proposal was to measure the complexity in the number of English words required to precisely define the memory model to a first year undergraduate student. For instance, sequential consistency can be explained with "memory as a table" abstraction, where the threads issue memory operations one at a time and these operations atomically read or update the table. Weaker memory models should seek for such simple explanations. On the other hand, an independent concern raised was for languages like C/C++ to allow programmers the ability to harness the raw power of the hardware, which falls very much in line with the philosophy of these languages. There was a lot of excitement about a proposal from Viktor Vafeiadis for a "rectified" C++ memory model that provides a logical foundation for solving the "out-of-thin-air" issues in C++ and Java. The model uses promises to model speculation and an execution is valid only when all threads eventually keep all of their promises. It also has the nice property that a restricted version of separation logic that only allows local reasoning is sound for this model. Another valuable direction of research is to identify the common patterns of relaxed variable usage in programs today to identify good patterns similar to "data-race freedom" which when enforced will provide easy-to-understand memory models.

5 Panel discussions

5.1 Panel Discussion: Concurrency vs. Parallelism

Matthew Sinclair (AMD Research – Bellevue, US)

This discussion session focused on parallelism and concurrency, both in practice and in how we teach it. Broadly, the discussion can be sub-divided into two areas of discussion: how and what we should be teaching parallelism to better equip our students for the post-university world, and what exactly is the difference between concurrency and parallelism.

Initially, the discussion revolved around teaching and using parallelism. In his talk prior to the discussion and again during the discussion, Mirko Rahn emphasized that many students are not adequately prepared to write (and debug) parallel programs in the real world. In particular, he highlighted three inter-related issues: lack of understanding about the cost of inter-processor communication, how long latency operations affect the order operations appear, and the cost of doing synchronization to ensure that operations appear in a consistent order across many processors. To Mirko, all of these problems are ultimately related to the latency of inter-processor communication, because it hurts performance, impacts what values a processor sees, and potentially requires additional, costly synchronization. Petr Kuznetsov pointed out that some of this boils down to how we teach students – and that we need to decide if we want to teach concurrency and parallelism or show students how it works. In his opinion, focusing on the underlying concepts is more important, which he does in a course he teaches that uses Java. However, Matt Sinclair pointed out that some of the universities are starting to introduce parallelism pervasively throughout the curriculum seem to focus more on the practical, performance-driven benefits. Michael Scott subsequently postulated that there is no one language or approach to teach these concepts, but that a steady, incremental approach is best – at earliest levels, we should introduce students to deterministic, independent parallelism (which have no subtleties); in later courses introduce more subtle but structured paradigms (e.g., messaging between distributed processors in distributed computing course; DRF model; event-driven programs in HCI courses); and finally in an OS course teach Peterson's algorithm (but only after making them comfortable with structured parallelism). Panagiota Fatourou presented an alternate, more theory-centric approach: she teaches an upper-level course on concurrency that starts with simpler concepts like mutual exclusion and uses small snippets of code to demonstrate how to write correct and incorrect code. Then she moves onto more complicated concepts, with liberal use of team parallel programming and examples to give the students practical experience.

The above discussion on parallel programming and concurrency led to a spirited discussion, initially raised by Victor Luchangco, about what exactly parallelism is compared to concurrency. Michael Scott emphasized that there is no one definition that separates them, and that reasonable people will disagree about them. Victor Luchangco and Michael Spear talked about concurrency in the context of robots (and later git repositories), as they believe that robotics represents one situation where students (especially students who are unfamiliar or less experienced with writing even sequential code) must think about events that are not dependent on one another (and thus could be done simultaneously). Michael Spears also focused on inexperienced programmers, and how they will likely only see a parallelized QuickSort algorithm as a performance optimization, not a way of thinking. In comparison, distributed systems (like robots) are much clearer about why concurrency is

necessary/important. Subsequently, Michael Scott opined that the key here is to present algorithms where control flow shows the potential parallelism – clear thinking is the most important part, not performance (which echoes Petr's point above). This struck discussion struck me as potentially the most interesting – even a room full of experts had problems coming to a consensus about what the definition of these terms are, as well as what the best ways to introduce them to students.

Participants

- Sarita Adve
University of Illinois – Urbana-Champaign, US
- Deepthi Devaki Akkoorath
TU Kaiserslautern, DE
- Hagit Attiya
Technion – Haifa, IL
- Naama Ben-David
Carnegie Mellon University – Pittsburgh, US
- Annette Bieniusa
TU Kaiserslautern, DE
- Hans-J. Boehm
Google – Palo Alto, US
- Irina Calciu
VMware – Palo Alto, US
- Nachshon Cohen
EPFL – Lausanne, CH
- Dave Dice
Oracle Corp. – Burlington, US
- Sandhya Dwarkadas
University of Rochester, US
- Panagiota Fatourou
University of Crete – Heraklion, GR
- Pascal Felber
University of Neuchâtel, CH
- Michal Friedman
Technion – Haifa, IL
- Tim Harris
Oracle Labs – Cambridge, GB
- Danny Hendler
Ben Gurion University – Beer Sheva, IL
- Maurice Herlihy
Brown University – Providence, US
- Antony Hosking
Australian National University – Canberra, AU
- Idit Keidar
Technion – Haifa, IL
- Alex Kogan
Oracle Labs – Burlington, US
- Petr Kuznetsov
Télécom ParisTech, FR
- William M. Leiserson
MIT – Cambridge, US
- Yossi Lev
Oracle Labs – Burlington, US
- Victor Luchangco
Oracle Labs – Burlington, US
- Virendra Marathe
Oracle Corp. – Burlington, US
- Maged M. Michael
Facebook – New York, US
- J. Eliot B. Moss
University of Massachusetts – Amherst, US
- Madan Musuvathi
Microsoft Research – Redmond, US
- Vijay Nagarajan
University of Edinburgh, GB
- Erez Petrank
Technion – Haifa, IL
- Michael Philippsen
Universität Erlangen-Nürnberg, DE
- Mirko Rahn
Fraunhofer ITWM – Kaiserslautern, DE
- Michael L. Scott
University of Rochester, US
- Marc Shapiro
University Pierre & Marie Curie – Paris, FR
- Nir Shavit
MIT – Cambridge, US
- Matthew Sinclair
AMD Research – Bellevue, US
- Michael F. Spear
Lehigh University – Bethlehem, US
- Viktor Vafeiadis
MPI-SWS – Kaiserslautern, DE
- Haris Volos
HP Labs – Palo Alto, US
- Heike Wehrheim
Universität Paderborn, DE
- Reinhard Wilhelm
Universität des Saarlandes, DE
- Felix Wolf
TU Darmstadt, DE
- Sebastian Wolff
TU Braunschweig, DE