The Velox Transactional Memory Stack

The transactional memory programming paradigm could become the coordination methodology of choice for actual and future multicore and many-core architectures. The transactional memory support spans a complete software and hardware stack, including programming language and hardware support, runtime and libraries, compilers, and application environments. The Velox project has developed such a comprehensive transactional memory stack.

The Velox Project
Advanced Micro Devices
Barcelona Supercomputing Center
Chalmers University of Technology
École Polytechnique Fédérale de Lausanne
Red Hat
Technische Universität Dresden
Tel Aviv University
Universitat Politècnica de Catalunya
University of Neuchâtel

The use of multicore processors, in which cores operate in parallel, each supporting multiple threads, makes the case for a broader use of parallel programming. The success of multicore architectures depends on software programmers' ability to harness parallelism within their software.

Traditionally, locks have served as the common coordination mechanism for concurrent programming. Unfortunately, lock-based programming is not a perfect solution. Coarse-grained locking is easy to program but scales poorly with the number of cores because of limited parallelism. Programs that use fine-grained locks can perform exceptionally well, but designing them is a difficult task better left to experts. Finally, even when programmed optimally for a given architecture, lock-based code might not scale when moved to machines with a different memory layout or more cores. We therefore need a new approach to multicore programming that retains scalability while preserving the ease of programming with coarse-grained locks.

Transactional memory could become the paradigm of choice for replacing or complementing locks in multicore programming. Transactional memory simplifies the parallelization of existing single-threaded code by eliminating the need to explicitly write fine-grained lock-based code. Programmers simply write code in which methods or blocks of code accessing shared data are declared high-level transactions, typically using transaction block language constructs. The synchronization and coordination details are left to the underlying transactional memory mechanisms.

Transactional memory executes in hardware, software, or as a combination of the two. Because transaction block constructs are typically provided at the programming-language level, transactional memory support spans the complete computer stack, from applications to hardware, encompassing language extensions, compilers, libraries, transactional memory runtime, and operating system. The Velox project (http://www.velox-project.eu) aims at building such an integrated stack for transactional memory.

Published by the IEEE Computer Society

```
Class Queue {
   QNode head;
   QNode tail;
   public eng(Object x) {
      __transaction {
        QNode q = new QNode (x);
        q.next = head;
        head = q;
      }
   }
}
```

Figure 1. Constructing a scalable concurrent first-in, first-out (FIFO) queue using transactions simply involves placing each method into its own atomic block.

A new paradigm for concurrent programming

Transactions let developers indicate that code sections must execute atomically—that is, as if they were run in isolation from the rest of the code. Transactional memory is a speculative execution mechanism in which accesses to shared objects can run simultaneously. In case of conflicting accesses, detected at runtime, one or several transactions might have to roll back and restart their execution. In other words, transactions execute optimistically in parallel and, as long as conflicts are rare, the performance gains resulting from the higher concurrency dominate the overheads introduced by transactional execution.

To illustrate why transactions are attractive from a software engineering perspective, consider the problem of constructing a scalable concurrent first-in, first-out (FIFO) queue that lets several threads enqueue items at the tail of the queue while other threads can dequeue items from the head of the queue, at least when the queue is not empty. Any problem so easy to state, and that arises so naturally in practice, should have an easily devised, understandable, and efficient solution. However, solving this problem with locks is quite difficult. In 1996, Michael and Scott published a clever but subtle solution.[1] The fact that solutions to such simple problems are considered difficult enough to be publishable results speaks poorly for fine-grained locking as the mainstream programming paradigm for the multicore era.

The VELOX Project Team Members

The VELOX consortium includes top research and system-integration organizations and transactional memory experts with nonoverlapping skills in the areas of computer architecture, operating systems, compiler and runtime systems, programming language design, programming models, and application and benchmark design. In addition to contact authors Pascal Felber and Etienne Rivière from the University of Neuchâtel, members of the VELOX team are:

- Walther Maldonado Moreira, Derin Harmanci, and Patrick Marlier from the University of Neuchâtel;
- Stephan Diestelhorst, Michael Hohmuth, and Martin Pohlack from Advanced Micro Devices;
- Adrián Cristal, Ibrahim Hur, and Osman S. Unsal from the Barcelona Supercomputing Center;
- Per Stenström from Chalmers University of Technology;
- Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka from École Polytechnique Fédérale de Lausanne;
- Vincent Gramoli from École Polytechnique Fédérale de Lausanne and University of Neuchâtel;
- Ulrich Drepper from Red Hat;
- Saša Tomić from Universitat Politècnica de Catalunya;
- Yehuda Afek, Guy Korland, and Nir Shavit from Tel Aviv University; and
- Christof Fetzer, Martin Nowack, and Torvald Riegel from the Technische Universität Dresden.

By contrast, it is almost trivial to solve this problem using transactions (see Figure 1). The solution consists of no more than placing the methods of a sequential queue implementation in a transactional block (`__transaction{}`). In practice, of course, a complete implementation would include more details, but even so, this concurrent queue implementation by itself is simple. Moreover, there is room for optimism: a FIFO queue implemented using a software transactional memory can deliver the same performance as Michael and Scott's hand-crafted fine-grained lock-based algorithm.

The VELOX transactional memory stack at a glance

As Figure 2 illustrates, the VELOX transactional memory stack consists of several basic components.

Language extensions and APIs are the most visible aspects of transactional memory for programmers. Although programmers could add transactional memory support to applications using explicit library calls or

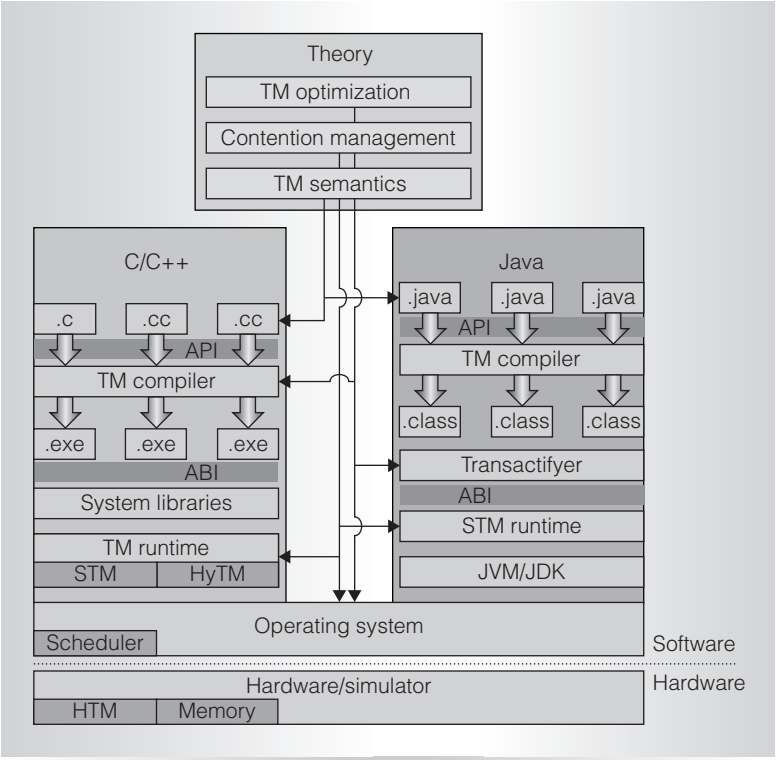

Figure 2. The VELOX stack supports two families of programming languages: C/C++ and Java and the associated tool chains and libraries. The VELOX project has studied a set of crosscutting research challenges to drive the design and development of the stack components. (TM: transactional memory)

declarative mechanisms (such as annotations), such an approach is not satisfactory for large systems. It relies on coding conventions, can lead to intricate code, and is often error prone. On the other end of the spectrum, automated source or binary code instrumentation work on simple examples, but are difficult to extend to realistic code. Adding new language constructs with well-defined semantics is the soundest approach for importing transactional memory support into existing languages.

Compiler support is necessary for implementing an efficient transactional memory stack in a way that is transparent to the programmer. For instance, it lets us identify transactional load and store operations on shared data and map them to the underlying transactional memory without requiring the programmer to explicitly mark these operations. Moreover, the compiler support lets us propose transactional memory-specific optimizations that improve the transactional code's performance and reduce the overheads of transactional memory accesses.

To better exploit transactional memory, system libraries must be adapted to execute speculatively inside transactions despite performing potentially unsafe operations (for example, I/O). Where applicable, a developer can also replace locks with transactions within libraries for better performance.

The transactional memory runtime is the transactional memory integrated stack's central component. It implements the transactional memory's synchronization logic. The VELOX project proposes several such libraries. A first class uses software transactional memory (STM) only; a second uses a mix of software-based transaction support with specific hardware extensions for scalability and efficiency; and a third relies on a pure hardware approach in which the transactional memory mechanisms are completely implemented in hardware (HTM).

Operating system extensions can help improve transactional memory's performance for some tasks relating to the system as a whole (scheduling, for example), in particular because transactional workloads coexist with traditional workloads. A typical example in the context of the VELOX stack is the support of transactional memory-aware scheduling in the Linux kernel to avoid preempting threads inside transactions and to serialize conflicting transactions on the same core.[2]

Finally, the complete stack builds upon a hardware platform that provides the necessary support for designing efficient transactional memory implementations. We use synchronization facilities such as Advanced Micro Devices' Advanced Synchronization Facility (ASF)[3] to support much of the application's transactional execution in hardware, and rely on software for unsupported transactions. Another approach is to design processors that fully support transactional memory in hardware.[4]

Other issues related to the VELOX stack's design and implementation include transactional models (for example, elastic transactions[5]), optimization strategies, scheduling algorithms, contention management, and progress and safety guarantees.

Language-level transactional memory constructs

Programmers have used transactional memory in the form of STM libraries for some time. However, when using these libraries, they must explicitly inject calls to the transactional memory libraries in their code (for example, for accessing shared memory locations). This method has severe disadvantages:

- It is harder to program and understand and is error-prone (for example, it's easy to forget to annotate a memory access).
- The compiler does not get insight into a transaction's actions and access patterns. The library cannot perform important optimizations such as detecting which data are shared or thread-private. In contrast, compilers can perform many transaction optimizations at compilation time.
- Close interaction between transactional memory and compiler-generated code such as exception handling is difficult to impossible.

Therefore, we have developed programming language extensions. For the VELOX project, we concentrate primarily on C/C++ and Java.

For both C and C++, the main extension is a construct to declare a transaction. In C, this is achieved with the `__tm_atomic` keyword, whereas C++ uses the `__transaction` keyword followed by a C++1x-style attribute, as documented in a draft specification (http://software.intel.com/file/21569). One can use this construct to introduce a transaction statement (that is, a block or expression). All memory accesses inside the lexical scope of the block or expression will be performed using the transactional memory library, unless they are proven to not conflict with any other thread's operations.

Figure 3 shows some of the C++ extensions. At the top, two declarations with attributes indicate whether the functions are transactional memory safe (that is, safe to be used in transactions). The following lines declare a transaction. The two declared functions are called within the transaction and then, depending on a condition, an exception might be thrown that cancels the transaction at the same time. This short sequence also shows how legacy code can be used. The `do_io` function is unsafe for use within transactions. The only prerequisite for using unsafe functions is that the transaction is marked with the `[[relaxed]]` attribute (which lets the compiler use a weaker kind of isolation).

```
[[transaction_safe]] void do_work () ;
[[transaction_unsafe]] void do_io () ;
__transaction [[relaxed]] {
  do_work () ;
  if (buffer)_full) {
    do_io (0) ;
    if (failure)
     __transaction_cancel throw "again";
  }
}
```

Figure 3. Transactional language constructs in C++. The `__transaction` keyword marks a transactional block. The associated attribute declares that the transaction shall use a weak kind of isolation to allow the use of transaction-unsafe operations such as `do_io()`.

The second noteworthy aspect of the work on the transactional memory language constructs is the integration of exception handling and transaction cancellation. Thrown exceptions can be declared to cancel the transaction (as in our example) or they propagate normally as in nontransactional code.

The resulting binary files use functions in a transactional memory runtime library. The ABI was developed to be compatible with other compilers and runtime from other interested parties in the industry (http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/#ABI). The goal is to avoid compatibility problems between different compilers and transactional-memory runtime implementations. The functions provided by the transactional-memory runtime enable optimizations that the compiler can implement, such as advanced read-set and write-set manipulations. The compiler can generate several variants of a transaction's code that will use different implementation variants in

the transactional memory library (for example, STM or HTM). At runtime, the transactional memory library chooses the implementation to be used to execute the transaction.

The C and C++ implementations share the transactional memory runtime ABI. We optimized the ABI for Linux and for x86/x86-64 architectures by reducing the overhead of the various calls and providing fast access to the thread-specific metadata required by most transactional memory runtime libraries. The VELOX stack implements transactional C/C++ extensions directly in the popular GNU Compiler Collection, as well as in the experimental Dresden transactional memory compiler (DTMC).[3]

On the Java side, we support transactions by adding an atomic block construct to declare transactions and a few extra keywords to handle operations such as explicit abort and retry. The VELOX project provides a transactional compiler, TMJava (http://www.tmware.org/tmjava), which processes Java source code with transactional constructs and generates pure Java classes that will subsequently be instrumented by the Deuce[6] framework to produce a transactional application.

Transactional memory runtime

We developed transactional memory runtimes for the two families of languages supported by the VELOX stack. For C/C++, we implemented a software-only and a hybrid runtime. For Java, our runtime executes on top of the Java virtual machine and is thus purely software.

TinySTM: An STM runtime

TinySTM is a lightweight and efficient STM implementation developed as the reference C/C++ transactional memory runtime of the VELOX stack.[7] It is word-based (that is, it achieves conflict detection at the memory address level) and uses revocable locks to protect shared data from concurrent accesses. It uses a single-version variant of the lazy snapshot algorithm (LSA)[7] and provides several operation modes—write-through or write-back updates with eager or lazy acquisition—as well as various contention managers.

TinySTM is purely implemented in software and supports the standard transactional memory library ABI (http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/#ABI). It performs and scales well on many types of workloads.[7]

ASF-TM: A hybrid transactional memory runtime

HTM achieves better performance than STM but has limitations (such as capacity) that prevent it from supporting all workload types. It is therefore typically coupled with an STM for executing transactions that cannot run entirely in hardware. In the VELOX project, we implemented a hybrid variant of TinySTM that uses ASF[3] as an HTM. As the simple example in Figure 4 shows, the transactional compiler transforms the original source code (Figure 4a) into an intermediate representation in which the `__transaction` block is replaced with appropriate calls to the transactional memory library to start (`_ITM_beginTransaction`) and to commit (`_ITM_commitTransaction`) a transaction (Figure 4b). It replaces reads (`_ITM_R*`) and writes (`_ITM_W*`) of shared data as well. We can link the resulting files against ABI-compatible transactional memory libraries.

The ASF-TM library implements the ABI by inserting the appropriate ASF instructions, as we describe later. Figure 4c shows the resulting code. Most operations are directly mapped to single ASF instructions. However, several features must be implemented by the ASF-TM software library. For example, if hardware transactions are aborted because of ASF capacity limitations, the library switches to a serial mode that executes one transaction at a time with no instrumentation. This software-only fallback mode is also used in other cases, such as to ensure that the transaction will eventually commit despite a high rate of conflict with other transactions.

Another challenge associated with ASF-based transactions is that they can be aborted at any time because of hardware conditions. Imagine, for instance, a hardware transaction that aborts while executing the `malloc()` function. This would leave the memory-management data structures in an

inconsistent state. Therefore, we developed variants that are robust to asynchronous aborts triggered by ASF for some functions.

Deuce: A Java transactional memory runtime

Deuce is an efficient open source Java framework that delivers full-featured transactional support to an existing application without changing its compiler or libraries.[6] This framework has two components:

- a transactification tool that processes annotated Java bytecode at load time to insert transactions and instrument loads and stores, and
- an STM runtime that executes transactions using one of the backend algorithms (currently, Transactional Locking II[8] and LSA[7]).

Deuce's original locking scheme detects conflicts at the level of individual fields, which provides finer granularity and better parallelism than object-based lock designs.

Deuce relies on Java annotations to specify transactions and so supports transaction blocks at the level of complete methods only. However, the TMJava front end compiles any Java source code with transactional constructs to generate the annotated bytecode that Deuce can use.

```
extern long cntr;
void increment () {
  __transaction {
    cntr = cntr + 5;
  }
}
(a)

extern long cntr;
void increment () {
  __ITM_beginTransaction (...) ;
  long 1_cntr = (long ) __ITM_R8 (&cntr);
  1_cntr = 1_cntr + 5;
  __ITM_W8 (&cntr, 1_cntr);
  __ITM_commitTransaction ();
}
(b)

; meml for cntr
SPECULATE
JNZ handle_abort
LOCK MOV RCX, 'meml
ADD RCX, 5
LOCK MOV meml, RCX
COMMIT
(c)
```

Figure 4. An example of how the transactional compiler transforms C++ code with a transaction statement (a) to target a transactional memory library application binary interface (ABI) (b) and to native code that uses Advanced Micro Device's Advanced Synchronization Facility (ASF) (c). For brevity, we omitted additional code around `speculate` for providing full semantics of `_ITM_beginTransaction`.

Hardware support for transactional memory

In general, HTM implementations are faster but are more difficult to design and verify than STMs. In the VELOX project, our aim has been to design transactional-memory hardware that remains simple yet offers high performance and scalability for key transactional memory applications. Meanwhile, VELOX STMs provide the ideal platform to test transactional memory extension ideas and to support common CPU architectures. These STMs must also provide good performance. We therefore exploit hardware-acceleration mechanisms provided by the processor, such as ASF, in the context of a hybrid transactional memory (HyTM).

EazyHTM

EazyHTM[4] is a scalable HTM implementation with eager conflict detection. EazyHTM lazily defers conflict resolution until commit time, and thereby avoids the difficulty of determining the most appropriate transaction to abort. In this approach, each transaction tracks conflicts with concurrently running transactions. Then, when a transaction reaches its commit point, it knows exactly which transactions need to be aborted to maintain system consistency. After all conflicting transactions have terminated, the transaction reaching its commit point publishes speculatively written values. Tracking all conflicts that a transaction encounters during execution can be useful. One direct consequence of this conflict map is that all nonconflicting transactions can commit fully in parallel.

We designed the EazyHTM protocol with existing chip multiprocessors in mind. It builds on commonly used directory protocols without requiring extensive changes. For example, EazyHTM does not require

splitting directories for parallelism or using a specific on-chip interconnection topology.

Advanced Synchronization Facility

ASF is an experimental AMD64 architecture-extension proposal for transactional programming and lock-free data structures.[3] It comprises seven new instructions:

- `speculate` and `commit` demarcate transaction boundaries;
- `abort` rolls back a transaction voluntarily;
- the `lock` prefix annotates transactional memory accesses (MOV);
- `watchr` and `watchw` add addresses to the transaction's read and write set; and
- `release` drops read-set addresses.

ASF's design focuses on implementation simplicity while maintaining a versatile, broad use of the mechanism. For simplicity, ASF reuses the existing, unchanged coherence and multiprocessor-interconnect protocols. Furthermore, it automatically aborts transactions that exceed ASF's limited capacity, are context switched, or are transitioning into the kernel.

Although ASF's actual capacity and execution behavior are closely tied to microarchitectural features, the specification ensures several baseline properties, which designers deemed as easily implementable:

- a minimal capacity that allows lock-free use cases without capacity-overflow logic in software;
- strong isolation, which protects transactions from other transactions and nontransactional code;
- general eventual-forward-progress properties that free the applications from worrying about repeated aborts due to difficult-to-control microarchitectural conditions (such as translation look-aside buffer misses or other resource conflicts);
- early abort, which prevents orphan transactions from continuing execution using stale data values.

Because ASF supports reporting of exceptions from within the transaction to the operating system, it can effectively handle transient conditions, such as page faults, ensuring eventual progress of the transactions.

ASF's remaining design choices—eager conflict detection, requestor-wins conflict resolution, and limited register checkpointing—follow the simplicity requirement.

VELOX evaluation of ASF

In the VELOX stack, ASF provides transactional memory support in the lowest layer. For its evaluation, we added several faithful, detailed implementations of ASF to a realistic out-of-order AMD64 processor simulator (PTLsim). The first implementation variant uses a new CPU data structure, the locked-line buffer (LLB), to monitor transactional memory addresses and keep backup copies of speculatively modified cache lines, which are restored if an abort occurs. We can configure the LLB to be of varying size; we evaluated eight- and 256-entry LLBs (LLB8 and LLB256).

The LLB is fully associative and thus is not susceptible to aborts due to associativity conflicts. However, it therefore has a limited capacity. Our second ASF implementation also uses the layer-1 (L1) data cache for read-set tracking. This implementation (LLB8-L1, LLB256-L1) can use the full L1 size as capacity for transactions, but the L1's limited associativity can reduce capacity for unfavorable address layouts.

These simulator-based implementations contain many of the microarchitectural subtleties and resulting performance and semantic implications found on current microprocessors and expected for future high-performance microprocessors, giving us high confidence that ASF is implementable in modern out-of-order cores.

To make ASF available for transactional-memory workloads, we developed a compiler tool chain that can generate code using the new ASF instructions. We also added software logic to our transactional memory runtime to alleviate ASF's hardware-induced limitations, such as limited capacity and system calls. In these unrecoverable and unretriable abort scenarios, we use a compiler-generated software fallback path that aborts all transactions and runs the transaction in serial mode.

Applications and performance

In the context of the VELOX project, we developed several applications to assess the transactional memory stack's benefits and benchmarks to evaluate its performance. We classify these programs into three categories:

- application use cases that demonstrate the usability of STMs;
- unit tests and microbenchmarks; and
- highly tunable realistic application benchmarks.

We compile these programs using the VELOX C/C++ and Java transactional-memory compilers.

Application use cases

We developed two multithreaded versions of the Quake game server that exploit STM. Atomic Quake[9] is derived from the parallel lock-based version of the Quake game server, in which atomic blocks replace all critical sections. This benchmark exhibits irregular parallelism, system calls, and error handling. Complex transactions include function calls, memory management, and nested transactions. To compare the programming effort with Atomic Quake, we created QuakeTM[10] from Quake's sequential version, keeping transactional memory-specific considerations in mind. Large atomic regions that put significant pressure on the underlying STM system characterize this application.

Among other C/C++ application use cases, we parallelized the Globulation2 real-time strategy game and developed a speculative stream-processing system. As for Java use cases, we implemented transactional memory-based Java beans for application servers.

Unit testing and microbenchmarking

To understand the performance and semantics of transactional memory runtimes, we designed TMunit,[11] which provides a simple and expressive domain-specific language for transactional memory workloads. TMunit allows expressing workloads as thread specifications consisting of a series of transactions that comprise, in turn, a series of read/write operations. TMunit is especially useful for unit testing because it runs replayable deterministic schedules of read/write operations, but can also test the performance of transactional memory runtimes in pathological scenarios or search for unnecessary aborts[12] as well as possible safety and liveness violations.

We also developed a set of transactional memory-based, lock-free, and lock-based concurrent data structures (hash table, linked list, skip list, and so on) to stress test the VELOX stack and identify performance bottlenecks with controlled workloads. These various data structures implementations let us demonstrate that STM scales well even without hardware support.[13]

Realistic application benchmarks

To evaluate the VELOX stack with sizable nondeterministic workloads, we developed the STMbench7 benchmark, which extends the well-known OO7 database benchmark. STMbench7 operates on an object-graph data structure with millions of objects and many interconnections between them. It can be configured to execute read-dominated, balanced read-write, and write-dominated workloads. We developed both a Java and a C++ version of STMbench7.[14]

The RMS-TM benchmark suite[15] includes lock-based and transactional memory-based implementations of seven real-world many-core applications from the emerging recognition, data mining, and synthesis domains that have a wide range of transactional memory characteristics in terms of transaction lengths, read/write set sizes, and contention. This benchmark suite is suitable for evaluating both STM and HTM systems. In addition, RMS-TM uniquely presents many desirable properties, such as nested transactions, I/O operations, and system calls inside transactions.

Evaluating the VELOX stack

Figure 5 presents some experimental results obtained with the VELOX stack. Figure 5a shows the STMbench7 running with the TinySTM runtime on a four quad-core AMD Opteron machine (16 cores). Although the read-dominated workload

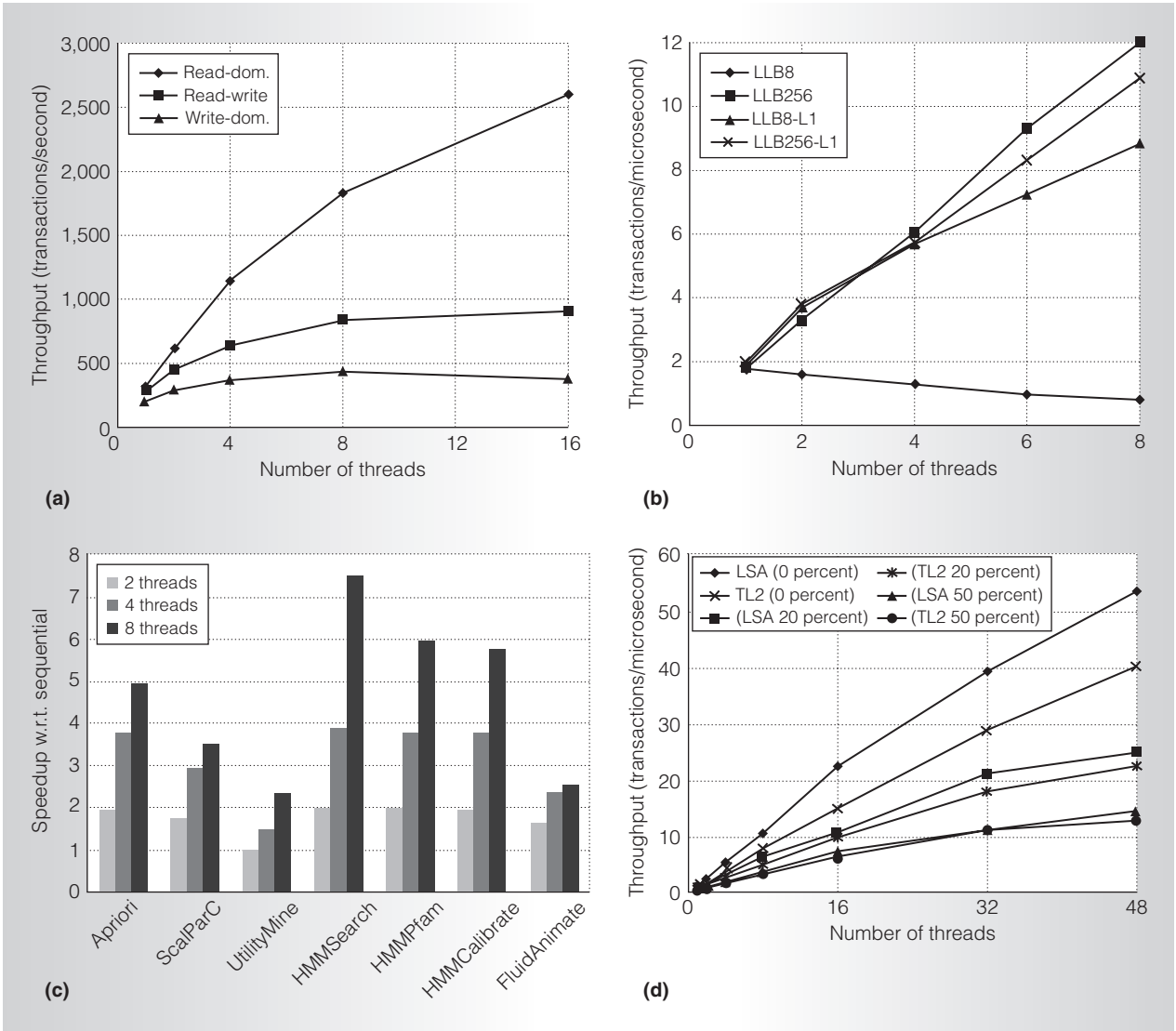

Figure 5. Performance of the VELOX stack: TinySTM runtime with the STMbench7 benchmark (a); ASF-based HyTM runtime with a red-black tree data structure (b); EazyHTM runtime with the RMS-TM benchmark suite (c); and Java-based Deuce runtime with a skip list data structure (d).

scales well, as we introduce more writes, performance flattens out with many threads.

Figure 5b shows the throughput of the ASF-based HyTM implementation on a red-black tree containing approximately four thousand elements with 20 percent update operations (insertions and removals). The LLB-8 implementation performs poorly because its capacity is insufficient for holding the parts of the data structure that are accessed, leading to constant execution of the software fallback path. LLB-256 scales remarkably well, and the L1/LLB variants perform only slightly worse because they are susceptible to cache-associativity limitations.

Figure 5c presents the scalability results of the RMS-TM benchmark suite using the EazyHTM implementation. As the graph shows, performance increases almost linearly with the number of threads on several benchmarks.

Finally, Figure 5d shows the throughput of the skip list microbenchmark in Java using the Deuce transactional memory runtime on a Sun UltraSPARC T2 Plus

multicore machine (two 8-core CPUs with eight hardware threads each). The list contains approximately 16 thousand elements, and we conducted experiments with three update ratios (0, 20, and 50 percent) and two STM algorithms (TL2 and LSA). Results demonstrate the good overall scalability of transactional memory with shared data structures.

We need more research to enable transactional memory as the abstraction of choice for leveraging the power of multicore and many-core architectures. We have identified a set of particularly important challenges, which our current and future work will address.

A premier challenge is the need for a larger and more diverse body of applications to be used as benchmarking and functional testing tools. This requires even more synthetical and well-understood benchmarks and larger-scale applications—either built especially for transactional memory or converted from existing massively multithreaded applications.

Use of transactional memory runtimes and libraries must follow the simplicity objective underlying the declaration of atomic regions to be executed speculatively, while letting the underlying transactional memory system handle the complexity. In contrast with this objective, transactional memory-unaware compilers require the programmer to instrument each transactional load and store. We therefore need transactional-memory-aware compilers, such as DTMC, or transactional memory extensions of existing commercial compilers. In addition to the simplicity aspect, transactional memory awareness at the compiler level allows for more optimization and lower runtime overhead.

The operating system also must be aware of the workload's transactional memory-specific aspects. For example, an important challenge lies in the interaction of the task scheduler and the transactional memory runtime to optimally schedule threads performing transactions (for example, by avoiding preempting threads inside transactions or by serializing conflicting threads on the same core to gain from locality).

Furthermore, the interaction between speculatively executed code and irrevocable actions such as I/Os requires being able to detect the occurrence of such actions and switch dynamically to an irrevocable mode.

Finally, a general challenge is associated with overall performance and spans the entire transactional memory stack. The overhead associated with transactional memory can currently degrade performance and therefore impact an application's scalability. MICRO

Acknowledgments

We thank Gina Alioto and Javier Arias for their support and involvement in the VELOX project. The research leading to the results presented in this article received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement No 216852.

References

1. M.M. Michael and M.L. Scott, ''Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms,'' *Proc. Symp. Principles of Distributed Computing* (PODC 96), ACM Press, 1996, pp. 267-275.
2. W. Maldonado et al., ''Scheduling Support for Transactional Memory Contention Management,'' *Proc. Symp. Principles and Practices of Parallel Programming* (PPoPP 10), ACM Press, 2010, pp. 79-90.
3. D. Christie et al., ''Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack,'' Proc. European Conf. Computer Systems (Eurosys 10), ACM Press, 2010, pp. 27-40.
4. S. Tomić et al., ''EazyHTM: Eager-Lazy Hardware Transactional Memory,'' *Proc. Int'l Symp. Microarchitecture* (Micro 09), ACM Press, 2009, pp. 145-155.
5. P. Felber, V. Gramoli, and R. Guerraoui, ''Elastic Transactions,'' *Proc. Int'l Symp. Distributed Computing* (DISC 09), Springer-Verlag, 2009, pp. 93-107.
6. G. Korland, N. Shavit, and P. Felber, ''Noninvasive Concurrency with Java STM,'' *Proc. Programmability Issues for Heterogeneous Multicores* (MultiProg 10), 2010; http://www.velox-project.eu/sites/default/files/multiprog10.pdf.

7. P. Felber et al., ''Time-based Software Transactional Memory,'' *IEEE Trans. Parallel and Distributed Systems,* preprint (16 Mar. 2010).
8. D. Dice, O. Shalev, and N. Shavit, ''Transactional Locking II,'' *Proc. Int'l Symp. Distributed Computing* (DISC 06), Springer-Verlag, 2006, pp. 194-208.
9. F. Zyulkyarov et al., ''Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server,'' *Proc. Symp. Principles and Practices of Parallel Programming* (PPoPP 09), ACM Press, 2009, pp. 25-34.
10. V. Gajinov et al., ''QuakeTM: Parallelizing a Complex Sequential Application Using Transactional Memory,'' *Proc. Int'l Conf. Supercomputing* (ICS 09), ACM Press, 2009, pp. 126-135.
11. D. Harmanci et al., ''Extensible Transactional Memory Testbed,'' *J. Parallel and Distributed Computing,* vol. 70, no. 10, October 2010, pp. 1053-1067.
12. V. Gramoli, D. Harmanci, and P. Felber, ''On the Input Acceptance of Transactional Memory.'' *Parallel Processing Letters,* vol. 20, no. 1, 2010, pp. 31-50.
13. A. Dragojević et al., ''Why STM Can Be More than a Research Toy,'' *Comm. ACM,* 2010, to appear.
14. A. Dragojević, R. Guerraoui, and M. Kapalka, ''Stretching Transactional Memory,'' *Proc. Conf. Programming Language Design and Implementation* (PLDI 09), ACM Press, 2009, pp. 155-165.
15. G. Kestor et al., ''RMS-TM: A Transactional Memory Benchmark for Recognition, Mining, and Synthesis Applications,'' *Proc. Workshop Transactional Computing* (Transact 09), ACM Press, 2009.

Pascal Felber is a professor of computer science at the University of Neuchâtel, Switzerland. He has a PhD in computer science from École Polytechnique Fédérale de Lausanne, Switzerland. He is a member of IEEE, the ACM, Usenix, and EuroSys.

Etienne Rivière is a researcher at the University of Neuchâtel, Switzerland. He has a PhD in computer science from the University of Rennes 1, France. He is a member of IEEE, the ACM, Usenix, and EuroSys.

Walther Maldonado Moreira is a PhD student at the University of Neuchâtel, Switzerland. He has an engineering degree in computer science from École des Mines de Nantes, France. He is a member of IEEE.

Derin Harmanci is a PhD student at the University of Neuchâtel, Switzerland. He has a MSc in computer science from École Polytechnique Fédérale de Lausanne, Switzerland. He is a member of IEEE and the ACM.

Patrick Marlier is a PhD student at the University of Neuchâtel, Switzerland. He has an engineering degree in computer science from the Université de Technologie de Compiègne, France.

Stephan Diestelhorst is a software engineer 2 at AMD's Operating System Research Center, Germany. He has a MSc in computer science from Technische Universität Dresden, Germany.

Michael Hohmuth is a member of the technical staff at AMD's Operating System Research Center, Germany. He has a Dr.-Ing in Computer Science from the Technische Universität Dresden, Germany. He is a member of the ACM.

Martin Pohlack is a senior software engineer at AMD's Operating System Research Center, Germany. He has a Dr-Ing in computer science from the Technische Universität Dresden, Germany. He is a member of EuroSys.

Adrián Cristal is a researcher and comanager of computer architecture for parallel paradigms at the Barcelona Supercomputing Center and the Artificial Intelligence Research Institute, Spanish National Research Council. He has a PhD in computer science from the Universitat Politècnica de Catalunya.

Ibrahim Hur is a senior researcher at the Barcelona Supercomputing Center, Spain. He has a PhD in computer science from the University of Texas at Austin. He is a member of IEEE and the ACM.

Osman S. Unsal is a group manager at the Barcelona Supercomputing Center, Spain. He has a PhD in electrical and computer engineering from University of Massachusetts, Amherst.

Per Stenström is a professor at Chalmers University of Technology, Sweden. He has a PhD in computer engineering from Lund University, Sweden. He is a Fellow of IEEE and the ACM and a member of Royal Swedish Academy of Engineering Sciences.

Aleksandar Dragojevic is a PhD student in computer science at École Polytechnique Fédérale de Lausanne, Switzerland. He is a graduate engineer in electrical and computer engineering from University of Novi Sad, Serbia.

Rachid Guerraoui is a professor at École Polytechnique Fédérale de Lausanne, Switzerland. He has a PhD in computer science from the University of Orsay, France. He is a member of IEEE and the ACM.

Michal Kapalka was a PhD student in computer science at École Polytechnique Fédérale de Lausanne, Switzerland. He has a PhD in computer science from École Polytechnique Fédérale de Lausanne, Switzerland.

Vincent Gramoli is a researcher at École Polytechnique Fédérale de Lausanne and the University of Neuchâtel, Switzerland. He has a PhD in computer science from the University of Rennes 1, France.

Ulrich Drepper is a consulting engineer at Red Hat. He has a MSc in computer science from Universität Karlsruhe, Germany. He is a member of IEEE.

Saša Tomić is a PhD student in computer architecture at the Universitat Politècnica de Catalunya, Spain. He has a MSc in computer science from the School of Electrical Engineering (ETF) in Belgrade, Serbia. He is a member of IEEE and the ACM.

Yehuda Afek is a professor at Tel Aviv University. He has a PhD in computer science from the University of California, Los Angeles.

Guy Korland is a PhD student at Tel Aviv University. He has a MSc in computer science from the Technion, Israel institute of Technology.

Nir Shavit is a professor at Tel Aviv University. He has a PhD in computer science from the Technion, Haifa. He is a member of the ACM.

Christof Fetzer is a professor at the Technische Universität Dresden, Germany. He has a PhD in computer science from the University of California, San Diego. He is a member of the ACM.

Martin Nowack is a PhD student at the Technische Universität Dresden, Germany. He has a MSc in computer science from the Technische Universität Dresden.

Torvald Riegel is a PhD student in computer science at the Technische Universität Dresden, Germany. He has a MSc in computer science from the Technische Universität Dresden.

Direct questions and comments about this article to Etienne Rivière, Computer Science Dept., Université de Neuchâtel, Emile-Argand 11, 2009 Neuchâtel, Switzerland; etienne.riviere@unine.ch.