SPECTRE: Speculation to hide communication latency

Jean-Philippe Martin, Christopher J. Rossbach and Michael Isard
Microsoft Research, Silicon Valley, CA, USA

ABSTRACT

We describe work in progress on the SPECTRE system which aims to provide high performance computing over distributed shared memory, targeting workloads such as graph algorithms for which functional or dataflow decompositions are inefficient. We exploit aggressive speculation to hide the latency of remote memory accesses and synchronization, and execute all code transactionally so that mis-speculations can be discovered and reverted. Unlike previous speculative transactional systems SPECTRE makes side effects visible beyond transaction boundaries before the transactions have committed, tracking dependencies to ensure correctness on abort: we call this property *transgression*. We outline the SPECTRE design and provide preliminary results from a microbenchmark to motivate the approach.

1. INTRODUCTION

This paper explores two linked hypotheses: that some algorithms are most naturally and efficiently implemented over a shared memory abstraction; and that speculation can be used to hide the communication latency that is frequently a bottleneck when implementing distributed shared memory.

We are building the SPECTRE system to test our hypotheses. It combines aggressive speculation with distributed software transactional memory to detect misspeculation and undo its effects. Our intention is to identify some performance-critical workloads that require more memory or processing power than is available on a single computer, but which benefit from the abstraction of mutable shared state. We will then compare their performance on our system with state of the art implementations on existing distributed execution engines such as MPI [18] and Dryad [12].

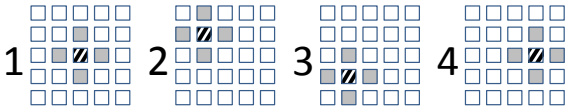

Figure 1: Four, numbered, steps of a Loopy Belief Propagation algorithm. In each step four values (grey background) are read while a single value (striped) is written.

Our focus on demonstrating the approach's potential leads us to building a runtime system first rather than an end-to-end programming model. The implementation and APIs of the system are designed to support high-performance general purpose distributed computing with speculation, and the applications it executes are ported by hand to exploit its features. We leave for future work some questions of developer simplicity, debugging support, and fault tolerance. The initial goal is to demonstrate that speculation can aid performance, and if this succeeds we plan to develop a layer that allows us to compile a high-level language directly to our runtime system.

The canonical type of algorithm that might benefit from SPECTRE is one that performs fine-grain updates to a large shared datastructure in an order that is difficult to predict statically. An example is sketched in Figure 1 which shows some steps in the execution of a Loopy Belief Propagation algorithm on a grid-graph: a small example graph is shown for clarity but a real application might span the memory of multiple cluster computers. In each step four values are read from the shared data-structure and used to compute a single result which is written back to the graph. The order of updates of graph nodes in this algorithm is a function of the previous updates so a strict functional decomposition introduces a dependency between each step; and since the ordering is data-dependent a static decomposition must conservatively insert a dependence between an update and the entire graph state at the previous step, which can be very inefficient. Many graph algorithms, including other machine learning and inference approaches, share this memory access pattern [16] as do standard problems such as mesh refinement.

In many practical cases like those shown in Figure 1

available parallelism is abundant, despite the fact that the algorithm is hard to express functionally. In the figure, the only constraint is that the operations in Step 4 must be executed after those in Step 1—Steps 2 and 3 could be executed in any order with respect to the other operations.

Speculation can help in two ways: first, some cores can run future steps in parallel, speculating that they do not need the output from the preceding steps. This allows a core to compute step 2 in parallel with step 1. Second, cores can advance through steps, speculating that what they have computed so far is correct. This allows a core to compute steps 1 and 4, consuming its own value without having to wait to hear whether the steps computed on other cores have modified what it is reading. In other words, speculation can discover parallelism and it can hide communication latency.

In order to maintain correctness it is essential to detect and undo mis-speculations. The engineering challenge we face is therefore to achieve performance gains from speculation that exceed the overhead introduced by bookkeeping and re-execution.

The SPECTRE system is currently a work in progress: we have a working prototype, but we are still exploring its design space, tuning its performance, and learning how best to program it. This paper therefore provides motivation and sketches the design, within the available space constraints, but reports only on provisional performance microbenchmarks.

2. COMPUTATIONAL MODEL

A SPECTRE program is made up of atomic code fragments called *tasks*. Instead of a total order as in a sequential program, one can specify a partial order of tasks to allow more parallelism. It can be represented as a *task graph*, as Figure 2 illustrates. If task b is reachable from task a in the task graph, then we write $a < b$. Our system allows a task to insert new successor tasks onto the graph during its execution, thus enabling iteration and recursion as in systems such as Cilk [1].

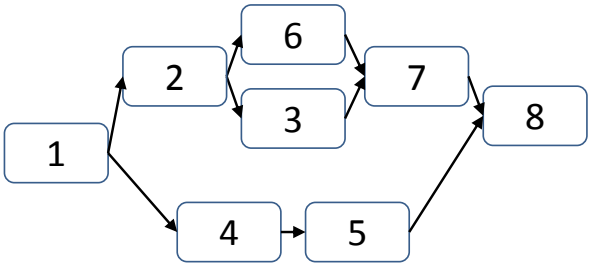

Figure 2: The structure of a Spectre program.

SPECTRE guarantees that the outcome of a correct program's execution is indistinguishable from a serial execution of the tasks that respects the partial order. The numbers in Figure 2 show one such possible order.

A task may receive read-only arguments when it is constructed, and all other inter-task communication is through a global set of shared objects. Each object has a unique identifier (OID), assigned at creation, which can be used as a reference to that object, for example by passing it as an argument to another task. Other side effects are possible, but for simplicity we focus on shared objects until Section 3.4.

The state of shared objects is managed using a transactional memory system in order to support aggressive speculation. Tasks may be executed concurrently or out of order, and consequent violations of the program serialization order are detected using the transactional memory. Mis-speculated tasks are then aborted and re-executed. For simplicity, and to maximize the opportunities for speculation, all code is executed transactionally.

In addition to adopting traditional software transactional memory techniques, SPECTRE makes side-effects visible beyond transaction boundaries when transactions have finished execution but not yet committed, a property we call *transgression*. This enables more aggressive speculation since there is no need to block computation while waiting for a distributed transaction commit. We track read dependencies between uncommitted transactions to maintain correctness in the face of transaction abort.

For the remainder of this paper, we use the term *transaction* to refer to an execution of a task. Each task may be executed multiple times, perhaps even concurrently, but at most one execution of a task will ever commit. The program completes when every task in the graph commits. We refer to transactions as though they were members of the partial order $<$. In addition we say a task a has committed if and only if any execution of a has committed.

3. SYSTEM DESIGN

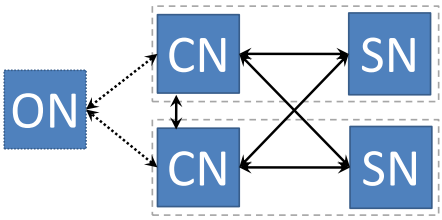

Figure 3: The Spectre system

The SPECTRE system (Figure 3) runs on a cluster of server computers and comprises a set of storage nodes (SN), a set of compute nodes (CN), and an ordering node (ON). These nodes are processes, and each computer may run more than one node: a typical deployment allocates an SN and a CN to every computer in the cluster. The SNs implement a distributed shared object space, the CNs execute transactions, and the ON coordinates task execution. The current implementation uses a centralized ON for simplicity, however we antic-

ipate this will become a bottleneck and a decentralized design to replace it is described in Section 5.

The SNs store the authoritative version of each object along with its version number. SNs can be queried for object versions, and they participate in the two-phase commit protocol described in Section 3.3. Each object is mapped to a unique, static SN. The SNs accept transactions in first-come, first-served order and have no knowledge of the task graph.

The ON stores the complete task graph, manages the assignment of tasks to CNs, and informs CNs when it is safe to commit a given transaction. When a SPECTRE application creates a task, it may specify a CN assignment for that task, or it can leave it up to the system to decide placement. Newly created tasks are sent to the ON, and it is informed when transactions commit.

Since the SNs are oblivious to the partial order, it is important to only commit tasks when all of their speculation has been resolved. To that effect, the ON checks when all of a task a's predecessors in the task graph have committed, and then it informs the corresponding CN that a is committable. A committable task might not actually commit right away (see Section 3.3). When a subgraph, all of whose predecessors have committed, lies entirely on one CN, that CN can make local decisions about commit order without communicating with the ON.

Each CN executes transactions, caches object values, and prefetches objects to make speculation more effective. SPECTRE makes use of concurrency both within a multi-core CN and across CNs. A single CN runs transactions concurrently on multiple cores, using transactional memory to manage conflicts. Each CN tracks the subset of the global task graph that has been scheduled locally, and detects local conflicts between reads and writes.

3.1 Task execution

Each CN concurrently executes as many tasks as there are cores on its computer, giving preference to transactions that are least likely to abort using topological order on the task graph. Where possible, SPECTRE avoids blocking computations while awaiting communication results.

A transaction can be in one of four states. It starts out *Running*, is *Completed* when all its code is executed, and ends up either *Committed* or *Aborted* depending on the outcome of the commit protocol. The CN tracks reads and writes to shared objects using a per-transaction *ReadSet* and *WriteSet*. A *ForkSet* tracks any tasks a transaction has created. Writes become visible to other tasks on the same CN when a transaction completes, and become visible globally when the transaction commits.

3.2 Object caching

Each CN caches multiple versions of objects, and speculatively uses cached values to satisfy a transaction's reads. Stale reads are detected by SNs at commit time, and a prefetching and best-effort invalidation protocol helps to keep cached objects up to date to avoid excessive aborts.

The task graph is used to select which version of an object the CN will use for transaction in a read request. This allows tasks to complete out of order: if $x < z$ and $z < y$ then z will read x's values even after y completes.

3.3 Aborting and Committing

If a transaction x aborts, the CN also aborts all transactions that read values written by x and schedules new instances of the aborted tasks. These actions can be decided locally: the ON only needs to be involved if x created new tasks that may have been dispatched to other CNs, in which case those tasks also need to be aborted.

If transaction x writes some value and then y reads it or overwrites it, this creates an ordering constraint: x must appear before y in the serialized order. The CN keeps track of these constraints as a dependency graph.

When a CN C hears from the ON that some transaction x is committable, it must wait until all of x's predecessors in the dependency graph have committed before it can attempt to commit x. Commit is implemented using a standard two-phase distributed protocol that includes all the SNs that store any object in the union of x's *ReadSet* and *WriteSet*.

3.4 Allowing side effects

In practice, a program must produce output beyond merely updating its own set of shared objects. We provide extensions to allow side effects that are familiar from other transactional memory systems. Specifically, SPECTRE supports commit and abort actions [11, 26]: When a transaction commits or aborts, its `OnCommit` or `OnAbort` method is called, providing the transaction with a mechanism to make side-effects visible or clean up after side-effects which must be reversed.

4. EVALUATION

While our system implementation is still a work in progress, we have implemented a full distributed prototype of SPECTRE and include one microbenchmark to motivate our hypothesis that transgressive speculation can lead to increased performance. Figure 4 shows timings for a benchmark that computes 800 elements of the series $x_i = x_{\lfloor i/8 \rfloor} \times x_{\lfloor i/9 \rfloor}$, where x_0 is a random 200×200 matrix. The results show that the overhead of our speculation is about 18%, and on this workload transgression is beneficial from two cores onwards. While it is premature to generalize this to real-world workloads and larger systems, we believe it demonstrates the promise of the approach.

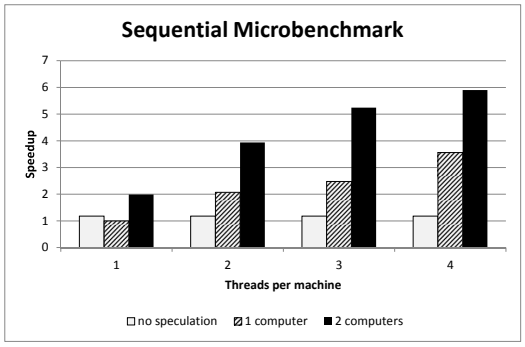

Figure 4: Speculation finds parallelism. Speedup is relative to the 1-node speculative case.

5. WORK IN PROGRESS

This section describes a number of features that we are planning to implement as we continue to develop the SPECTRE system. They are all motivated by our overall goal of supporting realistic workloads with high performance.

5.1 Advanced contention management

Currently, when we abort a completed transaction x we also abort any transactions that speculatively read values written by x. There are cases, however, where this is unnecessarily conservative: consider for example a transaction x that removes the head of a priority queue and passes it to some other transaction y that runs an expensive computation. x may have run speculatively out of order with another transaction z that modifies the priority queue, and so x must be aborted and re-run when z completes. If, on re-running x, the same object is at the head of the priority queue as before, there is no need to abort y and by tracking object values as well as their versions we can implement a contention-management policy that avoids the cost of re-execution in such cases.

5.2 Scheduling and locality

Co-locating computations with the SN that hold their data may be crucial for the performance of some workloads. We plan to investigate scheduling policies that leverage annotations placed on tasks describing the objects they are likely to read. There is a tradeoff between balancing load effectively and ensuring data locality, and it may turn out to be beneficial to allow object migration between SNs to improve overall performance. Annotations that predict access patterns can also be used by a CN to prefetch objects not already present in its cache, either before or during task execution.

5.3 Decentralized task graph

We anticipate that the centralized ordering node will become a bottleneck in our system. Although speculative computation can proceed on any CN while it is waiting for the ON to catch up with delayed commit notifications, deep speculation past commit entails overheads in the form of larger object version data-structures, and higher likelihoods of cross-CN conflicts yielding a higher abort rate. It is therefore natural to consider a decentralized implementation for the task graph in which the CNs communicate directly with each other about which tasks have committed and which are committable. With a decentralized graph, we may continue to use a centralized scheduler, or move to a work-stealing design like that used in the Cilk [1] system.

5.4 Conflict detection strategy

Our current design detects local conflicts at a CN when a transaction completes, and global conflicts when it attempts to commit. We plan to investigate alternative policies [17, 24, 25], to see which is best suited to our workloads and the distributed nature of the SPECTRE system. For example, an eager distributed conflict detection scheme allowing SNs to broadcast object writes to CNs may benefit some workloads by eliminating work wasted executing doomed transactions. Broadcasting all writes may prove too expensive, so annotations on objects or on writes may help the system prioritize update messages.

6. RELATED WORK

SPECTRE combines techniques from transactional memory and distributed shared memory. DSM presents a simple interface to mutable data, and STM is used to enable speculation in order to hide the latency resulting from synchronization and accessing remote data.

With relation to the other TM work, SPECTRE is an object-granularity [10], multi-versioned [3, 22] distributed STM system. SPECTRE has visible readers [23] and relies on eager conflict detection [17] at local nodes, while readers are invisible across CNs, and a lazy conflict detection [24] scheme is used to detect cross-machine conflicts based on commit-time validation of read-sets [6]. SPECTRE implements dependence-awareness [20, 21], allowing concurrent transactions on a single machine to share uncommitted state. A comprehensive review of the TM literature through 2010 can be found in [8].

Our emphasis on speculation guided our design. All-transactional programs have been proposed before [15], to simplify programming, but to the best of our knowledge, SPECTRE is the only TM design that allows the system to speculate past control flow boundaries between successive transactions. SPECTRE goes against received wisdom because it is not opaque [7]. Opacity requires that all transactions (not just the ones that will commit) see a consistent view of the memory. However, we do not believe this is a problem for SPECTRE because it executes all code transactionally, so the effects of errant transactions are never visible to non-transactional code. Conventional C# sandboxing is used e.g. to prevent native method calls from being based on inconsistent state, and doomed transactions are eventually discovered and aborted by our contention-management

system.

Mechanisms similar to those outlined in Section 5.1 have been developed in the past, for slightly different purposes: Galois classes [14] and transactional boosting [13] allow the programmer to provide inverse operations for concurrent data structures, in order to prevent false conflicts on operations that commute even though they may access some memory locations in common. Abstract nested transactions [9] do the same by isolating a computation. If the outcome is the same (despite false conflicts), then the rest of the transaction need not be aborted.

SPECTRE represents the program as a directed acyclic graph, like Cilk [1] which showed that distributed functional programs can be made to scale. Other systems such as TxCache [19] have explored transactional shared memory. We know of only a few systems that combine a transactional runtime with shared memory: ClusterSTM [2] shows that aggregating communication yields excellent scalability. Dash and Demsky's transactional DSM [4, 5] shows that prefetching and caching help mitigate the latency of distributed shared memory, and relies on transactional abort to recover from mis-speculation, as SPECTRE does. However they are not transgressive and wait to know that a transaction has committed before executing beyond it.

7. CONCLUSIONS

The SPECTRE implementation is still a work in progress. Our preliminary results convince us, however, that for inherently parallel workloads, transgressive speculation is a powerful tool to hide latencies generated by synchronization and approach an ideal parallel speedup.

8. REFERENCES

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPoPP*, 1995.

[2] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP*, 2008.

[3] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. In *OOPSLA*, 2005.

[4] A. Dash and B. Demsky. Software transactional distributed shared memory. In *PPoPP*, 2009.

[5] A. Dash and B. Demsky. Automatically generating symbolic prefetches for distributed transactional memories. In *ACM/IFIP/USENIX International Middleware Conference*, 2010.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, 2006. Springer-Verlag LNCS Volume 4167.

[7] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, 2008.

[8] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[9] T. Harris and S. Stipic. Abstract nested transactions. In *TRANSACT*, 2007.

[10] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, Jul 2003.

[11] O. S. Hofmann, C. J. Rossbach, and E. Witchel. Maximum benefit from a minimal HTM. In *ASPLOS*, 2009.

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[13] E. Koskinen and M. Herlihy. Concurrent non-commutative boosted transactions. In *TRANSACT*, 2009.

[14] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[15] B. C. Kuszmaul and C. E. Leiserson. Transactions Everywhere, 2003.

[16] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. *CoRR*, 2010.

[17] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *HPCA*. 2006.

[18] MPI. http://www.mcs.anl.gov/mpi/.

[19] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, 2010.

[20] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *MICRO*, 2008.

[21] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. In *PPoPP*, 2009.

[22] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*. Springer, 2006.

[23] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[24] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *ISCA*. 2008.

[25] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-lazy hardware transactional memory. In *MICRO*, 2009.

[26] C. Zilles and L. Baugh. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In *TRANSACT*. 2006.