Distributed Optimistic Concurrency Considered Optimistic

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

Abstract

Optimistic concurrency relies on speculative execution, read-write conflict detection, and checkpoint-rollback techniques to provide a programming model that replaces locks with the abstraction of atomic, isolated execution of critical sections. Previous research has shown that on chip multi-processors, a class of workloads featuring irregular parallelism and rare read-write conflicts can reap significant benefits from the TM model because complex synchronization code can be avoided without the scalability sacrifice that is the hallmark of coarse-grain synchronization. In a distributed *in a distributed setting*, however, with current technological parameters, this class of workloads becomes vanishingly small. Moreover, this class does *not* include the workloads currently used to evaluate TM and distributed TM systems.

We construct a model that predicts performance for a distributed software transactional memory (DSTM) executing a given workload. The model assumes optimal pipelining, batching, and locality, and predicts performance by finding the critical path induced by read-write sharing. We validate the model against real executions from TM benchmarks in the literature, finding that it tracks observed scalability to within 17%. We apply this model to popular TM benchmark applications, observing that none scale in a distributed context because transactions are too short in relation to network latencies. Traditional latency hiding techniques such as prefetching, batching, and speculation are do not help, and in fact, sometimes make performance worse. We conclude current TM benchmarks are not appropriate workloads for a distributed system using optimistic concurrency.

1 Introduction

Optimistic concurrency in general, and transactional memory (TM) in particular, have enjoyed considerable attention from the research community in recent years, due to their promise of a programming model that enables the developer to write scalable programs without writing complex synchronization code. This promise is very attractive for irregularly parallel programs, where available parallelism is abundant, but difficult-to-predict read-write sharing patterns make algorithms hard to express functionally. For such programs, a TM runtime system can discover ordering constraints dynamically using speculation, harvesting performance gains that exceed the overheads introduced by communication, accounting, and re-execution. In chip multi-processor environments, where a TM runtime enjoys low-latency communication through shared memory, TM's promise has been largely shown to be achievable [11]. Distributed software TM (DSTM) research [12, 13, 14, 9, 3, 5, 6] conjectures that similar performance improvements and complexity reductions can be achieved for irregular parallel programs at cluster scale.

This paper argues precisely the opposite: optimistic concurrency is unlikely to yield significant performance benefits in a distributed execution environment. DSTM is, in fact, *over-optimistic.* Transactional memory (TM) systems rely on speculation to hide synchronization latency induced by read-write shared objects or memory cells: in a distributed context, speculation may be further leveraged to hide communication latency that is frequently a bottleneck when implementing distributed shared memory. However, DSTMs can be effective at hiding these latencies *only* for workloads that have a profitable balance of computation latency to communication latency. When a workload cannot strike this balance, communication latencies cannot be hidden, compute resources must stall, and poor scalability is the consequence. The class of workloads for which a profitable balance be struck on modern hardware is vanishingly small, and notably, does not include the workloads evaluated in the TM and DSTM literature to date.

We build a model that predicts scalability for workloads on a general model of a distributed software transactional memory (DSTM) system. The goal of the model is to establish a bound on the im-

pact of read-write sharing on end-to-end execution latency for a given DSTM and execution platform. The model assumes optimal pipelining, batching of operations, and locality. We validate this model against real executions from the STAMP [22] benchmark suite, showing that it tracks observed scalability to within a 17% error rate when we model execution platforms matching those used to evaluate STAMP. We then synthesize models of multiple popular benchmark applications from the TM literature and observe that none scale well when we model platforms whose communication latencies approach those likely to be observed in a cluster environment. The poor scalability derives fundamentally from a poor balance of transaction execution latency and communication latency, which in turn derives from the sharing patterns in the benchmarks themselves. Moreover, traditional latency hiding techniques are ineffective. Batching transactions together reduces communication, but at the cost of creating additional conflicts that ultimately hurt scalability. Aggressive speculation past control flow boundaries [21] is similarly ineffective. We show that with today's technological parameters, current benchmarks are inappropriate workloads for a distributed STM: a modest-sized cluster (16 nodes) fails to scale past 2× even assuming RDMA-scale interconnect latencies, and all but one are slower than a serial all-local execution when communication latencies reach the order of TCP/IP message latency.

2 Models

To make our argument empirically, we must first construct effective models of the various artifacts present in a DSTM system. In this section we describe how we model a DSTM execution environment and implementation, and how we model the execution of a given workload within that environment. We elucidate a set of minimal assumptions about models of the platform, DSTM implementation, TM workloads, and show how wee can characterize and synthesize workload models that allow us to make performance estimates that have acceptable fidelity to real implementations.

The goal is to construct a model that applies to as many scenarios as possible. Since our goal is to estimate an upper bound on scalability, we can make simplifying assumptions that err on the side of reporting higher scalability: if we still observe that workloads hardly scale, then we know that the conclusion would still hold with a more accurate model. In this spirit we abstract away the communication needed to keep the shared memory consistent (e.g. two phase commit) and instead consider that if one machine writes to some memory that is the read by another, it takes at least one message delay for the data to go from one machine to the other. Our model therefore is that we have computers ("nodes") connected by a complete network with uniform latency and infinite bandwidth (so we use the same delay for all communication, without having to be concerned with compression techniques or Bloom filters). Memory is shared, and we represent the program to be executed as a partial order of tasks, representing all the allowable task interleavings. For example, if two functions A and B are run in parallel, then we place no edge between them in the program representation.

2.1 Workload model

A *workload* represents a set of possible executions of a particular program. Our goal is to estimate an upper bound on the scaling of that workload (the speedup relative to a single-computer execution). The workload (like the initial program) is a partial order of tasks, but in addition to the existing edges (that we call "control-flow edges"), we add new edges ("data-flow edges") to represent ordering constraints that arise through conflict. For example, if task A writes to some memory that task B then reads, then we insert a data-flow edge from A to B representing the fact that they cannot be run concurrently. These edges are inserted between any two tasks that access the same memory, when at least one of the accesses is a write. We omit these edges if the tasks are already ordered in the input program (doing so is safe since it only increases the scaling reported by the model).

We call the resulting graph the workload. Every total order that is compatible with the workload order represents a valid serialization of the program; those orders yield a correct output.

2.2 DSTM model

From the workload, we can compute the time to execute in a single machine ("serial runtime") and the time to execute on many machines ("parallel runtime"). Their ratio bounds the scaling available in that particular workload.

To estimate this time, we assign a computer and duration to every task. We also assign a delay for every data-flow edge that goes from one computer to another. The task duration is taken from the literature. The edge delay is set to the latency of a

communication link. We use several kind of links (from in-memory to TCP/IP) to determine the impact of this delay.

The sum of all node durations yields the serial runtime. The parallel runtime is determined by finding the graph's critical path and adding all delays along that path. Varying the number of machines yields different parallel runtimes. We use 16 when we compare to experiments with 16 machines (Section 4), and infinity when looking for an upper bound on available parallelism.

The parallel runtime does not take into account the time to detect conflicts and recover from them: instead we measure as if the scheduler knew in advance which tasks would conflict, and ran those sequentially to avoid having to re-execute anything. Once again this is safe because it only increases the reported scaling bound. Similarly, we are optimistic when accounting for synchronization: if task B reads what A writes, a real implementation might first send the data to an intermediate computer, or it may need several message delays to ensure all updates are atomic. Instead we count only a single message delay: this is a lower bound since the data at a minimum has to get from A to B.

The delay of committing changes is often a concern, as a single task may transactionally modify memory that is stored on multiple machines. Two natural optimizations are *batching* and transgression [21]; our model accounts for both. In the case of batching, the system commits multiple transactions in a single message, as it is faster to send a larger message than to wait for additional roundtrips. With transgression instead, the system commits transactions one at a time but it continues execution before hearing back, speculating that the transaction will commit. Those systems need extra bookkeeping to properly cascade aborts in case of misspeculation [30]. In our model, execution continues on the same machine without delay (just as it does with transgression). Across machines we only charge one message delay, as if each transaction committed immediately (as would be the case without batching). So in both case we are at least as fast as the best of both techniques. Misspeculation, in both cases, may cause multiple transactions to have to be re-executed. In our model instead we assume an optimal schedule that runs every task as soon as possible and avoids any re-execution. Thus our model subsumes batching and transgression: neither of these techniques should yield better scaling than we predict.

input	**description**	**source**
txlat	average transaction latency	emp
pcttx	% execution in transactions	emp
RS,WS	read-write set sizes	emp
rptx	retries per transaction	emp
wvcnt	number of waves	code
width	width of a *row*	code
depth	# of rows in a *wave*	code
forkjoin	fork-join connects waves	code
lanefreq	lane pattern	code

Table 1: Input parameters for the synthesis tool. Inputs marked "emp" are determined empirically (or from published data), while those marked "code" are determined by code inspection. A *row* is a set of *width* tasks that are all at the same depth in the task graph. We partition the execution into *waves*, each consisting of *depth* rows. Some waves use the fork-join pattern: single task join points precede and follow each wave. The *lane pattern frequency* indicates the number of control-flow edges: from 0 if each task has a single predecessor and successor to 100% when each is a successor of the whole previous row, and predecessor of the whole next row.

3 Methodology

In this section we describe the methodology used to construct the fundamental argument of the paper. In sum, to predict the scaling of an "ideal" implementation of a given TM benchmark, instead of trying to write such an ideal (D)STM to run the benchmark, we run the benchmark on a simple STM and then use the resulting trace to extrapolate its performance on an ideal (D)STM. We have implemented three benchmarks and the predictions match fairly well with the performance numbers that have been published before.

To enable predictions without having to reimplement each benchmark, we then built a trace generator that takes as input several parameters that describe the algorithm (determined from published numbers and after examining the benchmark's algorithm) and generates as output a synthetic trace. We can then run our tool on this trace to find the best-case scalability numbers.

Published benchmarks generally include sufficient information to estimate the following metrics: average transaction latency, percentage of total execution in transactions, and the average number of retries per transaction. We use these numbers in combination with the trace from our implementation of the benchmark to estimate best-case scalability.

We can also generate a trace without implementing the benchmark, by feeding these values below to our trace generator, along with additional parameters shown in Table 1. Many of the parameters are related to the construction of the task graph such that available parallelism in the modeled workload is accurately captured. Values controlling the structure of the resulting graph must be chosen based on code inspection of the implementation being modeled. The remaining parameters are used to synthesize read-write sets that feature sufficient read-write sharing of objects/cells to induce the required average number of retries per transaction at runtime.

3.1 Critical Path estimation

A DSTM hides communication latency by overlapping it with execution latency. To assess its potential, we estimate the runtime on a given cluster characterized by node count and inter-node communication latency, and for each benchmark we determine the minimum profitable average task latency *MPATL*, the minimum average task latency for the benchmark to finish more quickly on a parallel platform than when run sequentially (with no communication overheads). Given the actual average task latency *ATL*, their ratio is mathematically equivalent to the *average parallelism* metric described in [2], an upper bound on the scaling. We characterize it in terms of individual task latencies to emphasize the fact that scalability in a DSTM is a function of the amount of computation done per transaction. Both metrics require computation of the critical path through the graph.

The approach to computing these metrics is as follows. (1) Construct the workload trace, (2) Map tasks to nodes, (3) add dataflow edges only among tasks that may execute concurrently, (4) set vertex weights based on average task latency, (5) assign edge weights (dataflow costs one message latency, control zero), (6) find the critical path, (7) compute the minimum profitable task latency and estimate the runtime, taking the same approach as that in [2].

4 Evaluation

We have implemented three STAMP workloads for Spectre [21] (`yada`, `kmeans`, `vacation`). Based on traces from those implementations, combined with data from the STAMP IISWC paper [22], the critical path tool makes speedup estimates with a 28% geometric mean error from the numbers reported for the same workloads in the paper. We conclude that

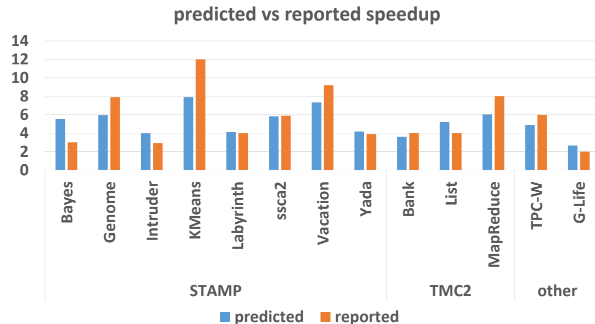

Figure 1: Validation of model based on synthesized workloads from STAMP [22], TMC2 [9], Volos et. al [32], and DiSTM [12].

our model predicts performance of real workloads to within a reasonable margin of error.

We collected data from benchmarks in the TM literature for which we could make reasonable estimates allowing us to characterize and model the workloads. Due to space constraints we do not evaluate the super-set workloads in the DSTM literature: many are inappropriate because they are synthetic [10], and in some cases, published data is insufficient to construct a high-confidence model. However, given the remaining list of TM benchmarks, we validate the model by showing that if we construct synthetic workloads based on the data in the papers, we accurately predict the performance reported in those papers. In fact, we are able to do this fairly well, with a geomean error of approx. 17% across all the benchmarks we consider, as shown in Figure 1.

Next, we take those same workload models and predict their performance across a range of communication latencies. We assume the communication cost in a hardware transactional memory (HTM) is approximately equivalent to a memory reference latency, or 0.000001 msec. We assume localhost communication latency (communication cost for a loopback-based STM implementation) is order 0.01 msec. We approximate an RDMA memory reference latency as 0.1 msec, and TCP-based cluster message latency as 1 msec. Figure 2 shows the predicted speedup of all the modeled benchmarks on a 16-node cluster, assuming these different inter-node latencies.

The result shows that even on a modest cluster (16 nodes), communication costs kill performance (see below). Even with RDMA-like latencies, few benchmarks are able to scale beyond 2x with 16 nodes, and all do better on a single machine with a similar num-

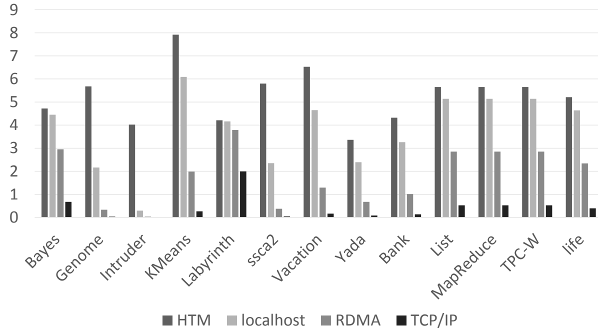

Figure 2: Speedup estimates for TM workloads from the literature on a DSTM platform across a range of message latencies.

ber of cores. Once TCP/IP latencies are involved, all but one workload is actually slower on 16 nodes than on 1. Moreover, the assumptions in the model are wildly optimistic about the ability of DSTM implementers to minimize communication. We introduce a single communication edge in the critical path IFF there is RW sharing between remote transactions: all other commits/validations and task management costs are free. In particular, if there is a sharing edge that is also a control flow edge, it is assumed to have zero latency: obviously this introduces error, but refining the model to eliminate the error would do nothing but strengthen our argument. The data below also assume ideal batching which is likely unachievable in a real implementation.

4.1 Limitations

While none of the workloads we looked at show scaling in a distributed setting at 16 nodes, it is possible that we have not considered some workloads that can in fact scale. Further exploration is required. We do not claim that the tasks done by the workloads cannot scale, just that the algorithms implemented appear not to. Different algorithms may scale better, and indicate interesting directions of study for abstractions for efficient distributed computation of irregular workloads.

5 Related Work

A comprehensive review of the TM literature through 2010 can be found in [11]. Trace-based performance estimation is based on real exections using Spectre [21].

We model TM workloads as a DAG of tasks, similar to Cilk [2], and rely on similar metrics: the ratio of *MPATL* to *ATL* is equivalent to Cilk's average parallelism, and our speedup estimates are derived from runtime estimates made using Cilk's techniques. We create models of workloads described in the TM and optimistic literature [9, 22, 32, 12, 14, 15, 10]. The critical path tool estimates available parallelism and scaling by relying on dynamic conflict information made available by instrumenting real executions [16, 25, 27, 26, 7, 1] or synthesizing workloads [10].

Programming models for shared mutable state Many systems have taken up the challenge of exploiting irregular parallelism [17, 19, 18, 24, 29, 4] leveraging workload properties such as commutativity [19, 31] or alternative programming models [20] and schedulers [23]. Other systems such as TxCache [28] have explored transactional shared memory. Combination a transactional runtime with distributed shared memory has enjoyed considerable recent research attention [12, 13, 14, 9]. Cluster-STM [3] shows that aggregating communication yields excellent scalability. Dash and Demsky's transactional DSM [5, 6] shows that prefetching and caching help mitigate the latency of distributed shared memory. Teraflux [20] aims to combine dataflow and task-based execution with transactional memory [8]. Shapiro *et al.* propose a variant of conflict-free replicated data types called *commutative replicated data types*, where all concurrent operations on that type commute [31]. Similarly, Conway *et al.* have added support for a limited form of aggregation to their BLOOM dialect of the Datalog logic programming language [4]. The combination of TM with a task-based programming model is explored in [8].

6 Conclusion

The class of workloads that can benefit from DSTM are performance-critical applications that require more memory or processing power than is available on a single computer, which benefit from the abstraction of mutable shared state, but which exercise that abstraction on conflicting data very rarely at runtime. Current workloads from the TM and DSTM literature meet only one of those criteria: the potential to reduce complexity with atomic operations over shared mutable state. Different workloads are required that are suitable for this kind of distributed execution, and our experience has been that workloads with the proper balance are very few and far between.

References

[1] M. Ansari, K. Jarvis, C. Kotselidis, M. Luján, C. Kirkham, and I. Watson. Profiling transactional memory applications. In *PDP '09: Proc. 17th Euromicro International Conference on Parallel, Distributed, and Network-based Processing*, pages 11–20, feb 2009.

[2] 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.

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

[4] N. Conway, W. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. Technical Report UCB/EECS-2012-167, EECS Department, University of California, Berkeley, Jun 2012.

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

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

[7] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[8] R. Gayatri, R. M. Badia, E. Ayguade, M. Luj$#225;n, and I. Watson. Transactional access to shared memory in starss, a task based programming model. In *Proceedings of the 18th international conference on Parallel Processing*, Euro-Par'12, pages 514–525, Berlin, Heidelberg, 2012. Springer-Verlag.

[9] V. Gramoli, R. Guerraoui, and V. Trigonakis. Tm2c: a software transactional memory for many-cores. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 351–364, New York, NY, USA, 2012. ACM.

[10] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM.

[11] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. 2010.

[12] C. Kotselidis, M. Ansari, K. Jarvis, M. Lujan, C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 51 –58, sep. 2008.

[13] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. C. Kirkham, and I. Watson. Investigating software transactional memory on clusters. In *IPDPS*, 2008.

[14] C. Kotselidis, M. Lujan, M. Ansari, K. Malakasis, B. Kahn, C. Kirkham, and I. Watson. Clustering jvms with software transactional memory support. In *IPDPS*, 2010.

[15] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[16] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *Proc. Symp. on Principles and practice of parallel programming (PPoPP)*, pages 3–14, New York, NY, USA, 2009.

[17] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *Proc. Symp. on Parallelism in algorithms and architectures (SPAA)*, pages 217–228, New York, NY, USA, 2008.

[18] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. *SIGARCH Comput. Archit. News*, 36(1):233–243, 2008.

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

[20] M. Lujan, I. Watson, and G. Roberto. Teraflux deliverable 3.3. Technical report, 2012.

[21] J.-P. Martin, C. J. Rossbach, and M. Isard. Spectre: speculation to hide communication latency. In H. Chen, Z. Zhang, S. Moon, and Y. Zhou, editors, *APSys*, page 18. ACM, 2011.

[22] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In D. Christie, A. Lee, O. Mutlu, and B. G. Zorn, editors, *IISWC*, pages 35–46. IEEE, 2008.

[23] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: a universal execution engine for distributed data-flow computing. In *PNSDI 2011*, Mar. 2011.

[24] D. Nguyen and K. Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *ASPLOS '11: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[25] K. Pingali, M. Kulkarni, D. Nguyen, M. Burtscher, M. Mendez-Lojo, D. Prountzos, X. Sui, and Z. Zhong. Amorphous data-parallelism in irregular algorithms. regular tech report TR-09-05, The University of Texas at Austin, 2009.

[26] D. E. Porter, O. S. Hofmann, and E. Witchel. Is the optimism in optimistic concurrency warranted? In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*. May 2007.

[27] D. E. Porter and E. Witchel. Understanding transactional memory performance. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Software Systems*, pages 97–108. March 2010.

[28] 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.

[29] D. Prountzos, R. Manevich, and K. Pingali. Elixir: A system for synthesizing concurrent graph programs. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '12, 2012.

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

[31] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Proceedings of the international symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 386–400, Grenoble, France, Oct. 2011.

[32] H. Volos, A. J. Tack, N. Goyal, M. M. Swift, and A. Welc. xCalls: safe I/O in memory transactions. In *EuroSys '09: Proc. 4th ACM European Conference on Computer Systems*, pages 247–260, Apr. 2009.