A Transactional Memory with Automatic Performance Tuning

QINGPING WANG, Lehigh University
SAMEER KULKARNI, University of Delaware
JOHN CAVAZOS, University of Delaware
MICHAEL SPEAR, Lehigh University

A significant obstacle to the acceptance of transactional memory (TM) in real-world parallel programs is the abundance of substantially different TM algorithms. Each TM algorithm appears well-suited to certain workload characteristics, but the best choice of algorithm is sensitive to program inputs, available cores, and program phases. Furthermore, operating system and hardware characteristics can affect which algorithm is best, with tradeoffs changing across iterations of a single ISA.

This paper introduces methods for constructing policies to dynamically select the most appropriate TM algorithm based on static and dynamic information. We leverage intraprocedural static analysis to create a static profile of the application. We also introduce a low-overhead framework for dynamic profiling of a running transactional application. Armed with these complementary descriptions of a program's behavior, we present novel expert adaptivity policies as well as machine learning policies that are trained off-line using simple microbenchmarks. In our evaluation, we find that both the expert and learned policies provide better performance than any single TM algorithm across the entire STAMP benchmark suite. In addition, policies that combine expert and learned policies offer the best combination of performance, maintainability, and flexibility.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent Programming Structures*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Atomicity, Serializability, Synchronization, Dynamic Adaptivity, Machine Learning

ACM Reference Format:
Wang, Q., Kulkarni, S., Cavazos, J., and Spear, M. 2012. A transactional memory with automatic performance tuning. ACM Trans. Architec. Code Optim. 9, 4, Article TBD (March 2012), 20 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Designers of general-purpose software components must strike an acceptable balance between maximizing performance in the common case and minimizing pathological problems in cases that are expected to be rare. As software complexity grows, this task grows increasingly challenging. The situation is particularly true for shared memory parallel programs, where architectural and workload characteristics dramatically affect the relative merit of various synchronization mechanisms.

In this paper, we focus on Transactional Memory (TM) [Harris et al. 2010]. There are more than a dozen software TM (STM) algorithms, each of which defines the common case differently: some are best for linked data structures, some for small operations on matrices, and others for read-dominated workloads. Some expect a strong language-level memory model, and some assume hardware will provide low cache miss latency and fast atomic instructions, such as compare-and-swap (`CAS`).

At Lehigh University, this research was sponsored in part by the National Science Foundation Grant CNS-1016828. At the University of Delaware, this research was sponsored in part by the DARPA Computer Science Study Group (CSSG) and National Science Foundation Career Award 0953667.
Author's addresses: Q. Wang and M. Spear, Computer Science and Engineering Department, Lehigh University; S. Kulkarni and J. Cavazos, Computer and Information Sciences Department, University of Delaware.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

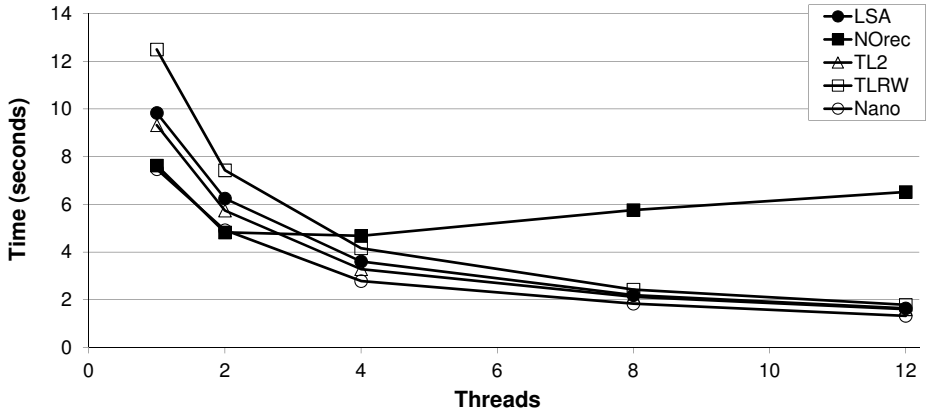

Fig. 1. Performance of the STAMP SSCA2 workload for common STM algorithms. The "Nano" algorithm is asymptotically worse than every algorithm tested, yet it is often best in workloads with short transactions.

From one iteration of an instruction set architecture to the next, hardware characteristics can change. Similarly, multi-chip systems behave differently than single-chip systems. Even on a fixed platform, behavior can vary due to sensitivity to program inputs [Hong et al. 2010] or phases of program execution [Shen et al. 2004; Lau et al. 2006]. As heterogeneity increases, the possibility of identifying any single TM algorithm as "best" grows increasingly unlikely.

We propose a comprehensive framework through which a TM runtime system can automatically tune its performance. In addition to measuring static properties of a transactional application, we introduce a dynamic profiling framework for TM. By combining these complementary approaches, we can develop rich descriptions of a workload without relying on complex analysis.

Armed with this information, we introduce expert heuristics and adaptivity mechanisms based on machine learning (ML). We take as inspiration applications of machine learning to solve systems problems [Stephenson et al. 2003; Kulkarni et al. 2004; Pouchet et al. 2008; Yotov et al. 2003]. Previous studies developed novel ML-based solutions for efficiently selecting compiler optimizations [Cooper et al. 2005; Agakov et al. 2006; Cavazos and O'Boyle 2006; Fursin et al. 2008], finding the best values for transformation parameters [Monsifrot et al. 2002; Stephenson and Amarasinghe 2005; Cavazos and O'Boyle 2005], and choosing the best algorithm to use for a particular sequential task [Li et al. 2004; 2005], to name a few examples. We extend the concept of ML-based adaptive runtime systems to apply to parallel programs and show three distinct ML techniques that can be employed in our general-purpose framework. Using a combination of static and dynamic information, our system can select among a broad set of TM algorithms during execution to select the algorithm most likely to maximize the performance of the in-flight program.

Our adaptive policies outperform any individual algorithm on the STAMP benchmark suite [Minh et al. 2008], and offer significant improvement when a strong language-level memory model is required. Our static analysis allows selection of high-performance special-purpose algorithms, and our dynamic mechanisms can exploit differences in program behavior over time to outperform any single algorithm for almost all workloads. As an example of the first property, consider Figure 1. The best performing algorithm is "Nano", a locking version of the original WSTM [Harris and Fraser 2003]. Nano is unsuitable for general-purpose STM workloads, as it has $O(reads^2)$ overhead, while the other algorithms have $O(reads)$ overhead. However, Nano has no metadata bottlenecks. For SSCA2's short transactions, our policies determine that Nano will provide the best performance.

This paper makes four main contributions:

— We present the first use of ML for runtime STM algorithm selection. Our system leverages static analysis and dynamic profiling, and considers strong and weak language-level memory models.
— Our best policies outperform any single algorithm across the entire STAMP benchmark suite.

— We identify static and dynamic characteristics (i.e., features) for the task of choosing the best STM algorithm for each phase of a program, and quantify their importance.
— Our technique can adapt to new architectures by simply retraining itself at install time.

Our best strategy tries to use expert knowledge to exploit static features; when that fails, it employs ML with simple dynamic features, such as cycles in transactions, cycles between transactions, and reads per transaction. On our test workload suite, a transparent case-based reasoning system was the best ML classifier, but black-box ML systems should perform well if they can avoid a few easily-identifiable mistakes. While almost every STM algorithm was useful in some scenario, choosing among eight algorithms ensured consistently good performance, suggesting that simple ML, simple features, and just a handful of STM algorithms should suffice in real-world settings.

The remainder of this paper is organized as follows. Sections 2 and 3 discuss the basics of STM operation and summarize prior adaptive STM systems. Section 4 introduces the static and dynamic features that we use to characterize workloads, and Section 5 describes our run-time adaptivity framework. Section 6 describes our expert and ML-based adaptivity policies. Section 7 presents performance results, Section 8 discusses future research directions, and Section 9 concludes.

2. STM BACKGROUND

To use STM, programmers annotate regions of code that require atomicity. Within these regions, individual loads and stores to shared memory are instrumented, as are region boundaries. Typically, the code within these regions runs speculatively: the instrumentation includes some mechanism to detect conflicts between concurrent transactions, as well as a mechanism to undo the partial effects of a transaction and retry it. The instrumentation for individual accesses and transaction boundaries is usually located in a library. The library provides concurrency control by mapping individual locations in memory to some form of metadata that enforces a single-writer, multi-reader protocol for program data. The library also handles conflict detection and rollback.

Below we list some key considerations for designers of STM algorithms:

— If aborts are rare, performing speculative writes *in place* may outperform buffering writes for commit-time replay. Note that this option requires transactions to maintain an "undo log" for reverting modifications in the event of an abort.
— Some algorithms offer low latency and high scalability when most transactions are read-only.
— Some workload/STM combinations are prone to performance pathologies and benefit from algorithms with provable livelock-freedom or fine-grained starvation avoidance mechanisms.
— The language-level memory model may require transparent "privatization" and "publication" to transition data between a state in which they are accessed within transactions and a state in which they are accessed nontransactionally. We consider ELA semantics, in which all forms of privatization are supported, but "racy" publication patterns are not [Menon et al. 2008].
— Hardware characteristics, such as the availability of vector instructions, the cache hierarchy, and the cost of `CAS` and memory fence instructions, affect tradeoffs among STM algorithms.

We now relate the above considerations to popular STM designs:

Single Mutex: All transactions are protected by a global mutex lock. There may be write logging to enable self-abort, and if read-only transactions can be statically shown to be common, a reader-writer lock may be used. I/O is always safe within a transaction, and there is no risk of livelock.

Ownership Records: Program data is mapped to a table of ownership records (orecs, essentially versioned locks) [Harris et al. 2010]. Reads are optimistic: they do not modify locks, but record lock versions. Writes acquire the lock either on first encounter or at commit time. Orecs allow for buffered updates or in-place updates with undo logging. Orec systems typically use some notion of global time (e.g., a shared counter) to reduce overheads. For workloads dominated by small writer transactions, this can become a bottleneck. Making an orec-based STM compatible with ELA semantics introduces significant overhead [Menon et al. 2008].

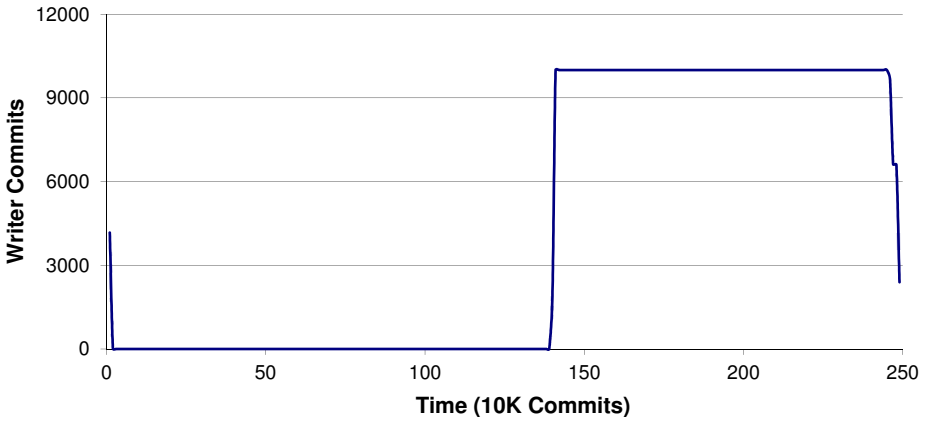

Fig. 2. Phases in the STAMP Genome benchmark. Each point on the X axis represents an interval of 10K commits, with the Y axis showing the number of commits in each interval that were not read-only.

Signatures: A transaction's accesses are represented as bit-vectors, or "signatures", and conflicts are detected by intersecting signatures. Example algorithms include the privatization-safe, livelock-free RingSTM [Spear et al. 2008], and the priority-focused InvalSTM [Gottschlich et al. 2010].

Values: Some STM algorithms log all address/value pairs that have been read. They can then detect conflicts by checking whether the values at these addresses have changed [Dalessandro et al. 2010], and use a single lock to serialize writer commits. These algorithms are livelock-free, and by virtue of maintaining no global metadata, they tend to have very low single-thread latency. They provide ELA semantics, but the single lock limits performance when writers are frequent.

Bit and Byte Locks: All of the designs mentioned above use optimistic read mechanisms, where no transaction can identify when it is accessing locations that another transaction is reading. By maintaining either bitlocks [Ni et al. 2008] or wider bytelocks [Dice and Shavit 2010], the cost of making transactional reads visible can be kept low. While reader visibility increases latency and can result in more contention for metadata, it simplifies conflict detection and resolution, enables ELA semantics, and simplifies support for advanced features.

Read-Parallel Designs: The eager and lazy TML algorithms are designed for workloads with infrequent writers. Latency is extremely low, at the expense of concurrency when there are writers.

To solidify our motivation for adaptivity, consider a program whose behavior is input-dependent and that operates in phases, where each phases' transactions exhibit characteristics for which a different STM algorithm is ideal. Such an application would suffer from selecting any single STM algorithm for its entire execution, even if the choice considered the input values and hardware/OS features. As a trivial example of this case, consider Figure 2. There is a distinct read-only phase in the application. An STM algorithm optimized for read-dominated workloads will be best for much of the execution, but a poor fit for the other phases.

3. PREVIOUS ADAPTIVE STM SYSTEMS

Past adaptive STM systems focused on preventing performance pathologies, with a few proposals also considering techniques to maximize performance. We highlight the most relevant works below.

Worst-Case Progress: Many STMs support a "serial-irrevocable" (SI) mode, where one transaction runs at a time. While SI was proposed as a way to support I/O in transactions that are known not to use self-abort, it can also guarantee progress. In essence, after a sufficient number of consecutive aborts, a thread may become serial-irrevocable (or perhaps only serial, if it might self-abort) to be sure that it will commit [Welc et al. 2008; Ni et al. 2008].

Location-Level Adaptivity: An STM can dynamically change the concurrency control mechanism for *individual variables*, allowing those involved in frequent conflicts to be accessed pes-

simistically [Sonmez et al. 2009]. This improves conflict detection and prevents some pathologies, without requiring pessimism on all accesses. Support requires overhead on every access to identify the variable's access mode.

Strong Progress Guarantees: Ni et al. proposed an orec-based STM that supports "obstinate" transactions (using visible reads) as well as switches to serial and serial-irrevocable modes. The system employed a novel indirection-based interface to prevent overhead while supporting these mechanisms, and avoided global coordination when switching the mode of a transaction: the instrumentation for any transaction seamlessly handled the fact of other transactions concurrently operating in other modes [Ni et al. 2008].

Feature Monitoring: ASTM [Marathe et al. 2005] tracked when a workload employed an uncommon API call ("early release") to decide whether locations should be locked on first access or at commit time. The technique increased throughput, but only if early release could be used.

Re-Parameterizing the STM: Dynamically selecting the number of orecs [Felber et al. 2008] improved scalability by decreasing the likelihood of false conflicts on metadata. Furthermore, workloads without much concurrency decreased latency by restricting themselves to a few orecs. An automatic mechanism based on re-execution found the best number of locks for a workload.

Phased Execution: PhTM [Lev et al. 2007] switched between hardware and software modes on a machine with hardware TM support. PhTM identified potential reasons to switch modes, including the presence of transactions that are unsupported by the hardware, excessive consecutive aborts, and periodic timers. Since its focus was on hardware/software interaction, PhTM did not consider switching among STM implementations. Some variants required coordination at the beginning of some transactions, even when there was no mode switch in progress. This is a potential bottleneck.

Selecting Locks or Transactions: Usui et al. employed static and dynamic analysis to identify workloads for which locks outperformed STM, even when multiple threads were available [Usui et al. 2009]. Clearly at one thread, the lower latency of a lock-based runtime is best. Additionally, if transaction latency is too high and the cost of a lock moving between processors' caches is low, the concurrency afforded by STM may not be worth its cost.

Pathology Avoidance: RSTM [Spear 2010] supports adaptivity among different STM algorithms. The system selects from 10 algorithms to react to bad performance. Decisions are based on likelihood of pathology and precision of conflict detection.

The above systems share the design philosophy that when throughput is unsatisfactory, the underlying TM becomes more pessimistic, according to a simple static sequence of transitions. Pessimism can take many forms: it can entail the use of commit-time locking instead of encounter-time locking, the use of coarse locks instead of transactions, the use of software instead of hardware transactions, or the use of visible reads. The pessimism may endure for only a few transactions, or may cause a permanent change to the underlying STM algorithm. Furthermore, the pessimism may be localized to one transaction's behavior, or may result in all transactions switching to a new STM algorithm. While many of the above mechanisms are robust and effective at preventing pathology, we found none to be suitable for our goal of maximizing system performance.

First, most mechanisms rely on the detection of bad performance by monitoring transaction aborts: if the consecutive abort count is too high or distinguished aborts occur (e.g., aborts due to overflow in hardware TM), then it is likely that adaptivity is necessary to restore progress. In the remaining mechanisms, detection of bad performance is done through analysis of a small feature set: early release in ASTM, transactional versus nontransactional time in Usui's system, and whole-program throughput in Felber's system. These limited feature sets are amenable to small adaptivity decisions (only one dimension in each system) or perhaps off-line decisions. We argue that to improve "already good" performance in a running application, a large feature set, consisting of both static and dynamic features, is necessary.

Second, the policies guiding transitions among algorithms are rigid. They are typically represented by a heuristic encoding an expert's intuition about how to prevent pathology for a specific STM algorithm. The danger in this approach is that it is not future-proof: the emergence of new hardware characteristics, new STM algorithms, and new workload behaviors can introduce situa-

Table I. Static workload features. Apart from `txsites`, values are `always`, `sometimes`, or `unknown`.

`S_txsites:`	Number of distinct source-level transactions
`S_nontx_gap:`	Function calls are made before or after the transaction, in the same lexical scope
`S_trivial:`	Transactions have no loops, no calls, and ≤ 6 reads and writes
`S_writer:`	Transactions always perform a write
`S_mcas_likely:`	Number of reads equals number of writes, and there are no calls within transactions
`S_txcalls:`	Transactions call functions that use TM
`S_nontxcalls:`	Transactions call functions that do not use TM
`S_verylarge:`	Transactions make at least 1 `nontxcall`, or at least 2 `txcalls`
`S_costly_aborts:`	Transactions make a `nontxcall` before any transactional reads or writes

tions that the expert did not anticipate. We argue that this diversity makes the creation of an optimal adaptivity policy intractable. Our solution is to create a system that can be trained in its deployed environment and potentially trained differently for each production application. In this manner, the STM system can automatically tune its performance based on its operating environment and changing workload characteristics.

4. CHARACTERIZING WORKLOADS

Any system that selects the best STM algorithm for a specific workload must have some description of the workload behavior that provides a reliable basis for decision making. As discussed in Section 3, past approaches used a variety of measures to approximate the behavior of workloads. Our approach combines information available through simple static analysis and dynamic measurement.

4.1. Static Features

We assume a standard TM interface: transaction boundaries are function calls, as are individual loads and stores; multiple transactions may be executed from within a single lexical scope; and functions that contain transactional instrumentation are marked. Our analysis counts distinct source-code transactions (`S_txsites`), approximates the distance between dynamic instances of these transactions (`S_nontx_gap`), and measures the incidence of various function calls within each transaction. A property is marked `always` if it holds for all transactions, `sometimes` if it holds for some transactions, and `unknown` otherwise. We do not distinguish between properties that do not hold and those that our analysis cannot prove to hold. Table I lists the static features we measure.

`S_nontx_gap` allows estimation of how likely transactions are to overlap. `S_trivial`, `S_writer`, and `S_mcas_likely` permit selection (or exclusion) of algorithms that apply to specific behavior patterns. The remaining features estimate the transaction's size: we assume that `S_nontxcalls` indicates large units of work for which transactional instrumentation would be prohibitively expensive, that multiple `S_txcalls` indicate data structure traversals, and that transactions that begin with a large prefix of nontransactional work merit special attention.

We emphasize that these features are easy to measure. More powerful analysis (e.g., shape analysis to determine the exact type of data structure being analyzed, or interprocedural analysis) are likely to enable more powerful adaptive policies.

4.2. Dynamic Features

To measure the dynamic behavior of a program, we employ lightweight instrumentation on every transaction boundary to measure program-wide properties. When additional information is needed, we use a simple STM (ProfileTM) to sample per-transaction characteristics.

Boundary Instrumentation: In the commit function of each STM algorithm, we update per-thread counts of writer and read-only commits. We query these counts when the workload read-only ratio (`D_RORatio`) is needed. After every commit we also store the value of the hardware tick counter, and before a transaction begins, we subtract that value from the current hardware tick counter and add the difference to a per-thread accumulator. Dividing by the number of transactions estimates the nontransactional work (`D_NonTxWork`) between transactions. In our environment, thread migration and frequency scaling did not affect the tick counter's precision.

Table II. Dynamic workload features measured by ProfileTM.

D_TxTime:	Cycles between start and end of a transaction; approximated via hardware tick counter
D_Writes:	Number of distinct locations to which a write is performed
D_WAWWrites:	Number of writes to locations that have already been written by the current transaction
D_ROReads:	Number of reads made by a transaction *before its first write*
D_RAWReads:	Number of reads to locations that have already been written by the current transaction
D_RWReads:	Non-RAW reads performed after a transaction's first write
D_RORatio:	Percent of transactions that are read only
D_NonTxWork:	Cycles between transactions within a thread; approximated via hardware tick counter
D_AllWrites:	Sum of D_Writes and D_WAWWrites
D_AllReads:	Sum of D_ROReads, D_RAWReads, and D_RWReads

Per-Transaction Sampling: Measuring dynamic properties of transactions at all times led to unacceptable overhead (more than 5% slowdown). Since profiling should have no cost when not in use, we instead use sampling. When a dynamic profile of the workload is needed, we switch to a custom STM implementation, which we call ProfileTM, and run several transactions.

In ProfileTM, a fair ticket lock guards all transactions. There is no concurrency, but profiled transactions are likely to be drawn from multiple threads. Since ProfileTM transactions are guarded by a ticket lock, they do not need to detect conflicts during execution. The removal of conflict detection reduces more latency than dynamic measurement adds, resulting in less single-thread latency than traditional STM algorithms. We use buffered writes and redo logs, since they simplify the task of counting the `D_WAWWrites` and `D_RAWReads` features discussed below. The features measured by ProfileTM are listed in Table II. We provide the motivation for each feature below:

— `D_TxTime`: Combining time inside of transactions with the always-measured `D_NonTxWork` indicates the frequency of transactions, as well as of the percentage of execution time attributable to transactions. This enables dynamic selection of locks instead of transactions.
— `D_Writes`: In STM algorithms that use per-location metadata (orecs, bitlocks, or bytelocks), each distinct address written requires a `CAS` operation, and thus the number of writes is a good indicator of overall transaction latency.
— `D_WAWWrites`: In undo-based STM algorithms, these writes are cheaper than `D_Writes`; in redo-based STM algorithms, these have the same cost as `D_Writes`.
— `D_ROReads`: A transaction often makes several reads before performing its first write. Most STM algorithms can capitalize on this behavior by offering a special read instrumentation that does not perform a test for read-after-write consistency. In redo-based systems, this is particularly effective, since it avoids a search in the write log.
— `D_RAWReads`: In redo-based STM algorithms, transactions that have performed a write begin each read operation by performing a lookup in the write log. When the lookup succeeds, the read immediately returns, resulting in very low latency.
— `D_RWReads`: In redo-based systems, these reads are most expensive, as they entail both a lookup in the write log and the overhead of a `D_RORead`.

In current STM algorithms, read-after-read is not detectable and offers little opportunity for decreasing latency. With aggressive compiler support, several additional optimizations are detectable. The most notable one is when a write-after-read can be detected by the compiler and transformed into a "read-for-write" operation. In systems supporting this optimization, a fourth read type would be needed. Similarly, a compiler might present a decoupled read instrumentation sequence [Harris et al. 2006] or offer special low-latency transactional reads that use partial sandboxing [Spear et al. 2009]. Little effort would be required to extend ProfileTM to count these accesses.

5. RUN-TIME ADAPTIVITY FRAMEWORK

We created the framework described in Figure 3 that can dynamically pick the correct STM algorithm during program execution using the workload features described in Section 4.

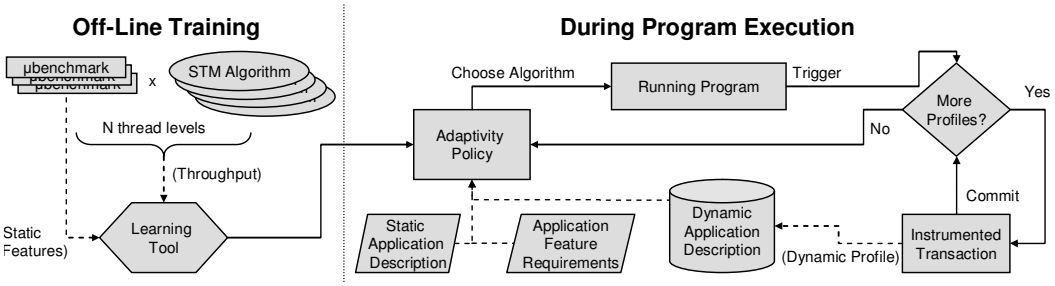

Fig. 3. Adaptivity Workflow: Off-line training on microbenchmarks produces an executable adaptivity policy. During program execution, various events (triggers) cause the framework to profile a fixed number of transactions, and then use the policy and profile to select a new algorithm. Algorithm selections also incorporate STM feature requirements, such as language-level semantics, and the static application profile.

5.1. Off-Line Training Strategy

We perform unsupervised off-line training. The trainer is given as input a set of microbenchmarks, a set of configurations of those microbenchmarks, and a set of STM algorithms. For each combination, it runs five 5-second experiments and records the average throughput. It then runs the experiment using ProfileTM in single-thread mode to gather dynamic characteristics of the workload and performs static analysis upon the microbenchmark. All data is fed to the ML training policy, which produces an adaptivity policy. This policy is either executable code or a data file that specifies the behavior of executable code, depending on the ML system being used.

There are two weaknesses in this approach. Our microbenchmarks (described below) are all homogeneous workloads with one program phase, which may not be representative of real TM workloads. Additionally, their coding style always matches the pattern:

```
transaction { tx_function(...) }
```

This pattern limits our intraprocedural analysis to the `S_nontx_gap` feature. As we discuss in Section 6, this artifact does not affect our expert adaptivity policies.

5.2. Off-Line Training: Workloads

In a production environment, one may tailor training data to the common-case for the target application. To show generality, we instead train using parameterized microbenchmarks, and thus measure what should serve as a lower bound on the effectiveness of our adaptive system. Our training workload consists of various configurations of the following microbenchmarks:

— **Data Structure Traversals:** Red-black trees, hash tables, and linked lists, with varying mixes of insert, lookup, and remove, and varying key ranges are stored in the dataset. These workloads typically scale well and correspond to the use of TM for creating concurrent data structures.
— **Pathology Test:** This usually causes livelock for eager STMs and starvation for lazy STMs.
— **Overhead Finders:** These expose overheads in STM algorithms. Examples include shared counters, which highlight boundary latency, truly disjoint workloads, which expose shared metadata bottlenecks, and read-sharing workloads, which emphasize the cost of visible reads.
— **Multiword Atomics:** These workloads use TM to perform multiword CAS operations of varying sizes or to implement read N write 1 operations. We also created a read N write N operation to show how the order of reads and writes affects throughput.
— **Database Simulations:** These workloads aim to mirror more complex uses of transactions. In addition to various "forest" workloads (consisting of multiple operations on a set of red-black trees), we also provide a tree workload where every transaction performs at least one write.

As appropriate, we varied the nontransactional time between transactions, the number of locations accessed within a transaction, and the percentage of transactions that were read-only. In total, this resulted in 213 different microbenchmark configurations, which we tested at many thread levels.

5.3. When to Trigger Adaptivity

To choose the best STM algorithm for a workload, the adaptivity policy must understand the workload's behavior. Past work focused on measuring the incidence of API calls that may never occur in the common case, motivating our use of dynamic profiling to measure characteristics that should apply to all workloads. During program execution, four events (triggers) activate our adaptivity framework. As in previous work, we set a threshold for consecutive aborts to rapidly detect pathologies. When a mutex-based STM is in use, there are no aborts, but long delays when attempting to begin a transaction may suggest that the algorithm should change. A second threshold watches this delay. Thread creation and destruction are also triggers, though the workloads we test do not exercise this feature. Lastly, we periodically resample a workload by using thresholds of total commits in the second thread.[1] The threshold values are $\{16^0, 16^1, 16^2, 16^3, k \times 16^4\}$, for all $k > 0$. All triggers are inexpensive and occur when a thread holds no locks.

5.4. The Dynamic Profiling Process

When configuring our library during program initialization, the adaptivity policy sets an initial algorithm, taking as input the required semantics of the application and whether the application uses self-abort. For the time being, we only select between Encounter-Time Lock Atomicity (ELA) [Menon et al. 2008] and weak semantics, which we sometimes refer to as "X" semantics. The policy is also given the application characteristics produced by static analysis.

On every trigger, the library blocks new transactions from starting and waits for all in-flight transactions to commit or abort. It then switches to ProfileTM and runs N transactions, one at a time. This ensures forward progress and prevents pathology. An important consideration is how the system should handle repeated recommendations of the same algorithm when consecutive aborts are frequent. Some workloads perform best with an algorithm that admits frequent aborts, and thus forbidding repeat selections is unacceptable. Instead, a repeat selection causes our system to record the total number of commits and aborts across all threads. On the next trigger, if the same algorithm is chosen *again*, then as long as there has been forward progress (e.g., more commits) under the chosen algorithm, it will remain in use, and the abort threshold for causing another trigger will be doubled. Commit-based triggers do not change the abort threshold.

6. ADAPTIVITY POLICIES

The system described in Section 5 allows many mechanisms for creating adaptivity policies. We envision three approaches: a programmer can create an "expert" adaptive policy, ignoring the entire left hand side of Figure 3; a completely automated ML system can generate the policy as the output of off-line training; or some guided process can be employed, wherein the programmer and learning tool create a policy together.

6.1. Expert Policies

These policies are written by a programmer to satisfy arbitrary requirements. For example, RSTM (upon which our work is based) provides state machines that avoid pathology by transitioning the algorithm to successively more pessimistic STM algorithms [Spear 2010].

Our simplest expert policies capture the intuition that the best algorithm depends on the thread count. We provide three policies, depending on whether ELA semantics are required or not and whether writers are expected to be frequent:

[1]Choosing the second thread prevents any triggers when an application runs in single-thread mode, without requiring overhead in all threads. For heterogeneous workloads, resampling parameters can be easily adjusted.

— **ThrX:** Assumes weak semantics are acceptable, and uses Mutex at 1 thread and the LSA algorithm otherwise. When ELA semantics are not required, LSA [Felber et al. 2008] is among the lowest latency and most scalable algorithms, unless contention is high.
— **ThrELA1:** Provides ELA semantics, using Mutex at 1 thread and NOrec [Dalessandro et al. 2010] otherwise. NOrec is among the most scalable STMs that provide ELA semantics.
— **ThrELA2:** Like ThrELA1, except for ≥ 8 threads, lazy TLRW [Dice and Shavit 2010] is used. TLRW has fewer bottlenecks than NOrec when writers are frequent.

When static and dynamic profile information is available, expert policies can become much more nuanced. Observing that many of the static features from Table I can also be detected via dynamic profiles, we identified several common use cases. When none of these cases is met, we fall back to ThrX or ThrELA1.

— **S_Trivial:** All transactions access fewer than 6 locations. If this is known statically, choose the Nano algorithm; if ELA semantics are required, choose TLRW with lazy acquire.
— **S_MCAS_Likely:** If the read and write counts are equal, the transaction is probably simulating a multiword-CAS. If the gap between transactions is large, use Mutex. Otherwise, favor an in-place algorithm (LSA or TLRW) appropriate for the required semantics.
— **RO:** If read-only transactions comprise more than 90% of all transactions, use TML.
— **Large:** If all transactions are large, choose NOrec at ≤ 4 threads, since its low latency for large transactions will outweigh its bottlenecks.

We refer to policies that select an algorithm using this set of rules as ExpertStatic, ExpertDynamic, or ExpertHybrid, depending on whether the characteristics are identified using static analysis, dynamic profiling, or both mechanisms.

6.2. ML-Based Policies

We consider three orthogonal techniques for automatically creating an adaptive policy through machine learning. These techniques all receive as input the same training data, but generate fundamentally different policies.

Case-Based Reasoning: In case-based reasoning [Aamodt and Plaza 1994] (CBR), a system creates a "case base" describing every program behavior that it observed during training, the environment (e.g., thread count and static features), and the best response (e.g., STM algorithm with the highest throughput). During program execution, CBR policies scan the case base for entries with the same number of threads as the workload. Among these entries, the policy selects the row that is most similar to the average of the collected transactional profiles and returns the algorithm named by that entry, which is the peak performer for some microbenchmark configuration.

Our CBR policies use the dynamic features described in Section 4. We consider all possible combinations as candidate similarity metrics, using a normalized Manhattan distance. By retaining some metadata in the case base, we can always identify the training experiment that influenced a CBR decision, which aids in performance tuning.

Neural Networks: In neural networks, training data is treated as k tuples, where the first field of each tuple is an output (o_k), and the remaining fields are a corresponding input vector (I_k). Through off-line analysis, the network learns a complex, high-dimension function that, for each vector I_k, computes the correct output o_k. The expectation is that if there is some mathematical relationship between program behaviors and the corresponding best choice of algorithm, then the network will learn that relationship and will be able to output the best choice for any new input vector. The most powerful neural networks are based on augmented topologies [Yao 1999], particularly the NEAT (Neuro Evolution of Augmenting Topologies) algorithm [Stanley and Miikkulainen 2002]. We used the open-source ANJI toolkit [Anji Home 2010].

Unlike CBR, NEAT is a black box classifier. We cannot precisely explain why some input produces some output. This property is both a strength and a weakness: while we cannot explain cases

where NEAT fails to identify seemingly obvious trends, NEAT has the potential to find relationships that are substantially more complex, nuanced, and robust than those found by CBR.

Rule Induction: An important benefit of applying ML successfully is a learning methodology that produces readable heuristics that gives the TM framework developer insight and confidence in their utility. Rule set induction is powerful learning methodology that can produce heuristics that are easy understand. Given the same training data as our CBR and NEAT policies, it produces a set of if-then-else rules. These rules are human-readable and show precisely what characteristics influenced the policy's decision. In large part, they resemble the structure (though not content) of our expert policies. For rule induction, we used the Ripper system [Cohen 1995].

7. EVALUATION

We built our adaptive policies in the RSTM framework [Spear 2010]. Our work included the addition of 7 new algorithms to the system. The 17 algorithms we evaluated are listed below. Algorithms marked with a "*" do not provide ELA semantics:

— Mutex: One lock protects all transactions. There is no concurrency, but latency is minimal.
— TML: A read-parallel STM where whenever a writer starts, all concurrent transactions abort.
— TMLLazy: A variant of TML where read-only transactions abort when a writer *commits*.
— LSA*: Ownership records (orecs) are used to detect conflicts, and writes are made directly to memory. Undo logs are used when a transaction aborts [Felber et al. 2008].
— OrecLazy*: LSA with commit-time locking and redo logs.
— OrecFair*: An extension of OrecLazy that adds starvation avoidance. "Possibly starving" transactions use visible reads, while others do not.
— OrecELA: An extension to OrecLazy that adds ELA semantics.
— NOrec: An STM based on value-based validation. There is no per-location global metadata, but transactions block during any writer commit [Dalessandro et al. 2010].
— NOrecPrio: Extends NOrec with a weak form of priority-based starvation avoidance.
— TL2*: A lazy orec-based algorithm [Dice et al. 2006] that achieves low latency by allowing some aborts that OrecLazy avoids.
— TLRW: Implements visible readers (read locking) via "bytelocks". Writes are performed in-place, with undo logs [Dice and Shavit 2010].
— TLRWLazy: A variant of TLRW with commit-time locking and redo logs.
— BitEager: Similar to TLRW, but using bitlocks instead of bytelocks.
— BitLazy: A bitlock-based TLRWLazy algorithm.
— RingSTM: To detect conflicts, readers maintain a signature of the addresses they access and at commit time writers publish a signature of the addresses they modify [Spear et al. 2008].
— TLI: A variant of InvalSTM [Gottschlich et al. 2010]. Committing writers are responsible for finding and forcibly aborting in-flight transactions with whom they conflict.
— Nano: A locking variant of WSTM [Harris and Fraser 2003]. Overhead is quadratic in the number of reads performed by a transaction, but there are no shared-memory bottlenecks.

All experiments in this section were performed on an HP z600 with 6GB RAM and a 2.66GHz Intel Xeon X5650 (Nehalem) processor (6 cores / 12 hardware threads). Code was compiled with g++ version 4.5.1, in 32-bit mode with –O3 optimizations. All experiments are the average of 3 trials. There was low variance among trials, except for the Bayes benchmark (see Section 7.4).

We trained 6 versions of our adaptive policies: "ELA" refers to training conducted using only algorithms that provide ELA semantics, and "X" refers to training on all 17 algorithms (e.g., weak semantics). We also considered three sets of training workloads from Section 5.2: S1 used data structure traversals, pathology tests, and overhead finders; S2 used multiword atomics and database simulations; S1+S2 used all training workloads. This led to 6 Ripper policies, 6 ANJI policies, and 6×31 CBR variants in our initial exploration (recall that with CBR, we must evaluate every combination of features separately). We set adaptivity triggers at 16 consecutive aborts, after a 2048-cycle loop spin on lock acquisition, and according to the commit thresholds described in Section 5.3.

Table III. Harmonic mean speedups (normalized to Oracle performance) for STAMP with no semantics requirements. Shown are the best single algorithm (LSA), the best expert policies (ThrX and ExpertStatic), the best Ripper configuration (trained on S1+S2), and the best CBR configuration (trained on S1, similarity based on weighted sum of D_TxTime and D_RORatio).

	Bayes	Genome	Intruder	KMeans		Labyrinth	SSCA2	Vacation		All
				(High)	(Low)			(High)	(Low)	
LSA	0.80	0.90	0.88	0.82	0.88	0.99	0.73	0.89	0.88	0.86
ThrX	0.80	0.94	0.98	0.89	0.92	1.00	0.79	0.97	0.96	0.91
ExpertStatic	0.72	0.95	0.98	0.98	0.99	0.96	0.96	0.96	0.95	0.93
Ripper	0.40	0.84	0.71	0.67	0.86	0.83	0.73	0.82	0.82	0.71
CBR	0.70	0.91	0.91	0.78	0.87	1.05	0.93	0.99	0.99	0.89

Table IV. STAMP Harmonic mean speedups when ELA semantics are required. ThrELA2 replaces ThrX, and CBR achieves its best performance using the D_AllReads feature.

	Bayes	Genome	Intruder	KMeans		Labyrinth	SSCA2	Vacation		All
				(High)	(Low)			(High)	(Low)	
NOrec	0.88	0.91	0.89	0.65	0.72	0.99	0.56	0.88	0.88	0.79
ThrELA2	0.92	0.90	0.90	0.72	0.73	0.99	0.67	0.86	0.88	0.83
ExpertStatic	0.76	0.92	0.93	1.00	0.92	0.95	0.90	0.94	0.95	0.92
Ripper	0.46	0.90	0.90	0.73	0.86	0.83	0.90	0.94	0.95	0.79
CBR	0.79	0.99	0.91	0.84	0.87	0.93	1.05	0.98	0.99	0.92

On any trigger, we collected a single transaction profile, as initial studies did not find a significant improvement in sample quality, but did observe noticeable slowdown in the Labyrinth workload, when collecting multiple profiles. Tuning this parameter based on `S_txsites` is future work.

We configured our Expert policies to detect behavior using only static STAMP features, only dynamic features, or a combination (static features + dynamic read-only ratio). We refer to these variants as ExpertStatic, ExpertDynamic, and ExpertHybrid.

7.1. Evaluation Criteria

To evaluate our adaptive policies, we used the STAMP benchmark suite [Minh et al. 2008]. For the 9 recommended configurations[2], we tested each of the 17 STM algorithms at 1, 2, 4, 8, and 12 threads. Using this information, we created an "oracle" dataset consisting of the best performing STM algorithm for each benchmark at each thread level.

For each adaptivity policy, we tested each benchmark at each thread level and computed its speedup versus the oracle. We scored each policy based on its per-benchmark harmonic mean speedup as well as its STAMP-wide harmonic mean speedup.

7.2. Performance Summary: Preliminaries

Tables III and IV list the best per-benchmark and STAMP-wide harmonic mean speedups for each adaptive system. Note that the oracle policy differs between the two tables, since ELA excludes several algorithms. Quantitative comparisons cannot be made between tables.

If only one algorithm can be used for all of STAMP, ELA favors NOrec while LSA is best otherwise. However, for several benchmarks this choice is far from ideal, resulting in a low 0.79 overall speedup for NOrec, and 0.86 for LSA. For X semantics, only TL2 was close to LSA (0.81); for ELA, TLRW and orec variants were close to NOrec (above 0.73).

The adaptivity policies included in RSTM perform poorly (not shown). These policies interpret transient high abort rates as pathologies and make permanent decisions toward fair but low-throughput algorithms. NOrec and LSA outperform the corresponding ELA and X RSTM policies.

Similarly, ANJI performance was unacceptably low. As a black-box classifier, our only recourse was to alter the training workload suite and re-train. This process never produced a policy capable of surpassing 0.57 speedup for X semantics and 0.70 for ELA semantics.

[2]We omitted the "yada" benchmark, since the released code is unstable.

7.3. Outperforming the Oracle

An ideal adaptivity policy should be able to outperform the oracle, i.e., achieve a speedup > 1. In practice, the incidence of speedup > 1 benefits from four program characteristics.

(1) The program should have distinct phases. The phases can be statically identifiable, or correspond to a dynamic property, such as the contents of a worklist [Kulkarni et al. 2009].
(2) Each phase must be long enough that the cost of profiling and then switching to a different algorithm can be offset by an increase in performance.
(3) Each phase must vary in behavior relative to other phases, so that the use of different algorithms for different phases can result in statistically significant improvements in performance.
(4) Ideally, phase boundaries should coincide with behavioral changes that are quickly detected by our triggers and that are understood by our adaptive policies.

Past work has shown that phases are not always clearly delineated (e.g., in the Lee-TM routing workload [Ansari et al. 2008] upon which STAMP Labyrinth is based, the abort rate "creeps" over time). In this case, the first and fourth properties are likely to be violated. However, the use of periodic profiling can still detect phases, so long as some other artifact (in Lee-TM/Labyrinth, the number of reads and writes) differs from one profile collection to the next.

Across all adaptivity policies, we found that at least one of our policies was able to achieve a consistent speedup of 1.04 or higher across all thread levels on each of KMeans, Vacation, SSCA2, and Labyrinth. However, these policies did not necessarily perform well on other workloads, and hence may not be reported in Tables III and IV.

Failure to outperform the oracle on Genome was a surprise. Upon further investigation, we found that some of our policies failed to choose TML when Genome entered the read-only phase, and others failed to abandon TML when Genome left the read-only phase. These faults can be addressed either by modifying TML so that departing the read-only phase causes a high rate of aborts, or else by adding a trigger based on the number of consecutive read-only or writer commits. Determining which approach is better should be delayed until there are more transactional workloads.

7.4. Expert Policy Performance

The ThrX and ThrELA policies, which select an algorithm based only on the thread count, raise performance significantly. For ThrX, this improvement is completely due to avoiding overhead at 1 thread, as it chooses LSA otherwise. We recommend this approach without hesitation for any future STM design. However, ThrX still performs poorly on SSCA2, KMeans, and Bayes.

ThrELA2, which chooses between Mutex, NOrec, and TLRWLazy, is more nuanced. In choosing TLRWLazy at ≥ 8 threads, it loses performance on Vacation. However, TLRWLazy scales better than NOrec for small writing transactions, and in the end this improvement on KMeans and SSCA2 tips the scales in favor of ThrELA2 over ThrELA1 (which only uses Mutex and NOrec).

It is worth noting that KMeans and SSCA2 transactions match simple patterns that can be detected statically: the S_mcas_likely and S_trivial patterns, respectively. The ExpertStatic policy, which has access to the static analysis of each program, can thus choose Nano for SSCA2 under X semantics (TLRW at ≥ 2 threads under ELA) and Mutex for KMeans with 2–4 threads. These choices have a striking impact on overall performance. ELA performance improves from 0.83 to 0.92. Under X semantics, performance rises from 0.91 to 0.93.

Apart from Bayes, ExpertStatic performance is excellent. While we include Bayes performance in our evaluation, we are generally suspicious of this workload. The number and size of transactions run by each thread is dependent on the interleaving of a few transactions executed early in the workload; eager STM algorithms (particularly with visible reads) seem to cause a bad initial commit order, which can cause an order of magnitude slowdown. Similarly, a round-robin scheduling of transactions can occasionally cause a superlinear ($> 4\times$) speedup at 2 threads.

Lastly, we observe that ExpertStatic outperformed ExpertDynamic (where conditions like S_trivial and S_mcas_likely are detected statically) and ExpertHybrid (ExpertStatic + the dynamic

read-only ratio). In the latter case, adding read-only ratio did not affect how we adapted, but added computation during adaptivity. In the former case, there was a conflict between the accuracy of our profiles and the aggressiveness of our policies. For example, ExpertDynamic never chooses Nano, because the cost of choosing Nano incorrectly is significant, and dynamic profiles are an approximation. This is particularly true for heterogeneous workloads: Bayes, Labyrinth, and KMeans all have both tiny transactions and large transactions for which Nano is unacceptable.

7.5. Ripper Performance

Our ML policies based on rule induction did not offer a significant advantage over using a single algorithm or the ThrX/ThrELA policies. Ripper output is human readable, and there is much room for an expert to fine-tune Ripper output.

Our analysis identified two factors that reduced performance. First, Ripper repeatedly chose a bad fallback algorithm: if no rule was invoked, it selected the low-performance TMLLazy algorithm, where LSA and NOrec would have been better choices. Second, and more significant, is variance in Ripper's ability to deduce ranges. For example, if two training experiments showed that TLRW was best at 2 threads with 8 `D_ROReads` and at 2 threads with 12 `D_ROReads`, Ripper should output a rule of the form $8 \leq D\_ROReads \leq 12$, but sometimes produces $D\_ROReads \in \{8, 12\}$. The latter is clearly less general. We are currently investigating quantizing strategies for the training workloads, so that the framework can specify ranges, rather than relying on Ripper to learn them.

With these limitations, Ripper's best "X" policy only reached 0.71 speedup (0.78 with Bayes excluded), and under ELA semantics, reached as high as 0.79 (0.87 without Bayes). This is significantly better than ANJI and competitive with ThrELA if we exclude Bayes. With a richer interface between our framework and Ripper (especially by quantizing profiles within the framework) as well as better static analysis of microbenchmarks, Ripper performance should improve.

7.6. CBR Performance

We explored all combinations of CBR features and considered all three training workload sets. Given this large search space, we were able to find policies that offered strong performance on STAMP, even without static profiles. With ELA semantics, the use of a single feature, D_AllReads, achieved a 0.92 speedup. This surpasses all other ELA adaptivity policies. With X semantics, our best performing policy only reached 0.89. This policy combined `D_TxTime` and `D_NonTxWork`, each of which independently achieved a 0.89 performance.

Table V shows the effectiveness of each individual dynamic feature on each STAMP benchmark, for both ELA and X semantics. The predictive power of each feature on a workload is clearly dependent on the set of available algorithms. For example, using D_AllReads outperforms the oracle on SSCA2. In contrast, time-based metrics (D_TxTime and D_NonTxTime) provide a better metric for the Labyrinth workload. We caution the reader against placing too much emphasis on these specific results, as the interplay between the set of available algorithms and the similarity of training workloads' transactions to test workloads' transactions is very nuanced. Still, there are clear and logical trends. For example, under X semantics the best performing algorithm often requires a CAS per location written, and we see that D_AllWrites is more predictive than D_AllReads. Under ELA semantics, the best algorithms either have a single CAS regardless of the number of writes (NOrec) or have very high overhead on each read (TLRW and BitLock variants). In these cases, D_AllReads is more predictive.

7.7. Impact of Training Data

In Figure 4, we show the effect that different training data has on the effectiveness of our best CBR policies and on our Ripper policy. Our CBR policies without exception performed best when trained only on the S1 training workloads. Ripper, on the other hand, showed different preferences under ELA and X semantics. Note that in the training workloads, S1 is drawn from STM microbenchmarks, whereas S2 models behaviors that we expect of future TM programs.

Table V. Performance of individual dynamic features for CBR adaptivity policies on STAMP. Column labels of "X" and "E" correspond to weak and ELA semantics, respectively.

	D_AllReads		D_RORatio		D_TxTime		D_NonTxTime		D_AllWrites	
Training Data	X	E	X	E	X	E	X	E	X	E
Bayes	0.69	0.79	0.73	0.03	0.83	0.88	0.84	0.94	0.65	0.81
Genome	0.89	0.99	0.83	0.67	0.93	0.82	0.93	0.91	0.91	0.95
Intruder	0.95	0.91	0.91	0.89	0.90	0.95	0.86	0.90	0.85	0.95
KMeans (high)	0.62	0.84	0.95	0.97	0.82	0.73	0.83	0.73	0.83	0.73
KMeans (low)	0.71	0.87	0.94	0.96	0.80	0.75	0.84	0.77	0.87	0.80
Labyrinth	0.92	0.93	0.84	0.62	1.05	1.06	1.04	1.08	1.06	0.99
SSCA2	1.06	1.05	0.89	0.90	0.92	0.74	0.89	0.90	0.90	0.89
Vacation (high)	0.98	0.98	0.96	0.68	0.90	0.95	0.88	0.95	0.99	0.92
Vacation (low)	0.96	0.99	0.96	0.73	0.94	0.97	0.93	0.96	0.89	0.94
All	0.84	0.92	0.88	0.18	0.89	0.86	0.89	0.89	0.87	0.88

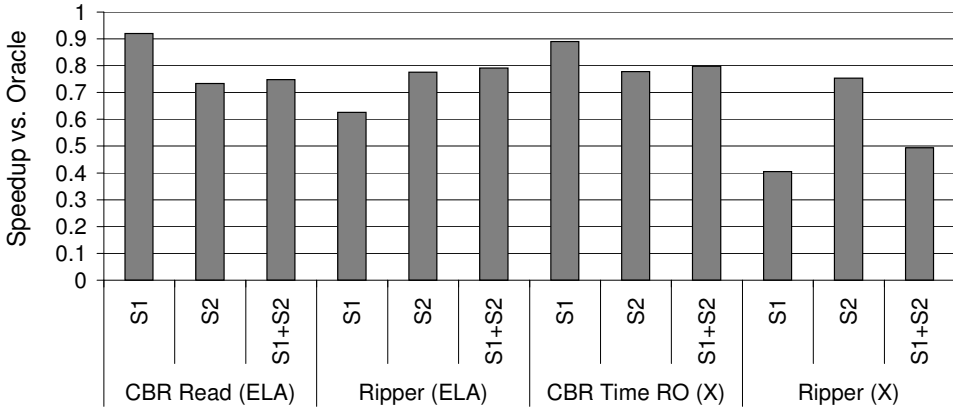

Fig. 4. Impact of training data. The best-performing CBR policies degrade significantly when trained improperly.

For CBR, the explanation is simple: S2 contains many entries that, on a per-metric basis, are indistinguishable to our CBR similarity functions (which simply match patterns). Thus the S2 workloads can cause our policies to reject an otherwise valid choice of algorithm from S1, due to a similarity collision. In the cases where S2 data led to a better decision than S1, it was by a small margin, whereas when S2 led to a worse decision than S1, it was by a large margin.

Ripper does not rely on pattern matching, but analyzes the training data to produce rules describing how features should influence adaptivity. The rules produced from the S2 data were typically long (15 different *if* statements, with 1–4 conditions per statement), whereas the rules produced from the S1 data were short (5 *if* statements). In both cases, Ripper chose the low-performance TMLLazy algorithm when all other conditions evaluated to false. By leading to generation of more opportunities to pick something other than TMLLazy, the S2 data set produced better policies.

7.8. Evaluation of Learning Features

By evaluating each combination of features from within the CBR classifier, we gained a sense for their predictive power. Table V provides a summary. The results illustrate the importance of dynamic profiling, since D_AllReads, D_AllWrites, and D_TxTime are features collected by ProfileTM. However, individual features have high variance, depending on semantics and training data.

As discussed previously, our training harness was not compatible with our static analysis, since all transactional code was reached via virtual dispatch from within a single source-level transaction. However even the performance of ExpertStatic (Tables III and IV) shows the effectiveness of the static features: SSCA2 exhibits the `S_trivial` property, KMeans exhibits the `S_mcas_likely` property, and Labyrinth exhibits `S_costly_aborts`.

Table VI. Ripper rules for weak semantics. Rules were produced from static features, using leave-one-out cross validation on STAMP. "ST" is an abbreviation for "sometimes".

Bayes, SSCA2	Genome, KMeans	Intruder, Vacation	Labyrinth

if (S_costly_aborts == ST)
  return NOrec
else return LSA
	if (S_costly_aborts == ST)
	  return NOrec
	else if (S_txsites >= 15)
	  return OrecFair
	else return LSA
		if (S_costly_aborts == ST)
		  return NOrec
		else if (S_txsites >= 15)
		  return OrecFair
		if (S_txsites >= 1
		  return Nano
		else return LSA
			if (S_txsites >= 15)
			  return OrecFair
			else if (S_txsites >= 10)
			  return Nano
			else return LSA

Table VII. Ripper rules for ELA semantics. Rules were produced as in Table VI.

Bayes	Genome, SSCA	Intruder, KMeans, Vacation, Labyrinth

if (S_txsites >= 10)
  return BitLazy
else if (S_txsites >= 5)
  return TLRW
else return NOrecPrio
	return NOrecPrio	if (S_trivial == always)
		  return BitLazy
		else return NOrecPrio

To gain further insight, we performed leave-one-out cross validation on the STAMP benchmark suite, by training a Ripper policy for one benchmark using the performance and static features of the other 6 benchmarks as inputs. Tables VI and VII show the resulting rules (note that at 1 thread, Mutex is always used). While `S_trivial` and `S_costly_aborts` appear in the Ripper output, emphasis is placed on `S_txsites`, the number of source-level transactions. Furthermore, while LSA and NOrecPrio are good final choices, Nano could be chosen inappropriately. In practice, the policy always chooses LSA for X semantics, achieving 0.91 speedup. For ELA semantics, the policy chooses between BitLazy and NOrecPrio, attaining 0.80 speedup.

7.9. Combining ML with Expert Policies

Throughout the conduct of this research, we were tempted to exploit expert knowledge, rather than allow our ML systems to operate as "black boxes". Section 7.8 provides an example of the cost: the inclusion of Nano in our training data resulted in its selection at inappropriate times. Experts would not expect Nano to be a good choice simply because there are between 10 and 14 source-level transactions. Experts would expect Nano to be a good choice only if *every* source-level transaction within a program phase is known to be trivial. The reasonable performance of the cross validation Ripper policy was pure luck, in that Nano was never selected in our experiments.

Special-purpose algorithms like TML and Nano are hard for a general-purpose ML system to understand. To show a classifier that the algorithms are worthy of consideration, one must train on workloads where the algorithms perform well. This, in turn, leaves the classifiers free to attribute the success of those algorithms to the wrong features.

However, these special-purpose algorithms are easy for experts to exploit. Our expert policies combine two approaches: they first identify a set of easily detectable cases, for which the expert knows exactly how to achieve peak performance. Two prominent examples are using TML when the read-only ratio is above 95%, and using Nano when all transactions are statically known to be extremely small. At some point, the cases became too hard to specify, and the "best" algorithm for a case became little more than a guess. At this point, the expert policies adopt a one-size-fits-all approach, by falling back to ThrX or ThrELA.

In Section 7.7, we showed that our CBR policies performed best when their training workloads did not exhibit the `S_mcas_likely` and `S_trivial` features (set S2). Even with these features in the training set, our CBR policies never achieved 0.95 speedup or higher for KMeans, SSCA2, or Labyrinth, the STAMP benchmarks that possess these features. In contrast, our ExpertStatic policy consistently performed well on these workloads, since the features are easy for an expert to exploit.

We now discuss policies that combine expert knowledge and ML-based learning. In these policies, the expert specifies cases that they can easily detect, and for which they know exactly which algo-

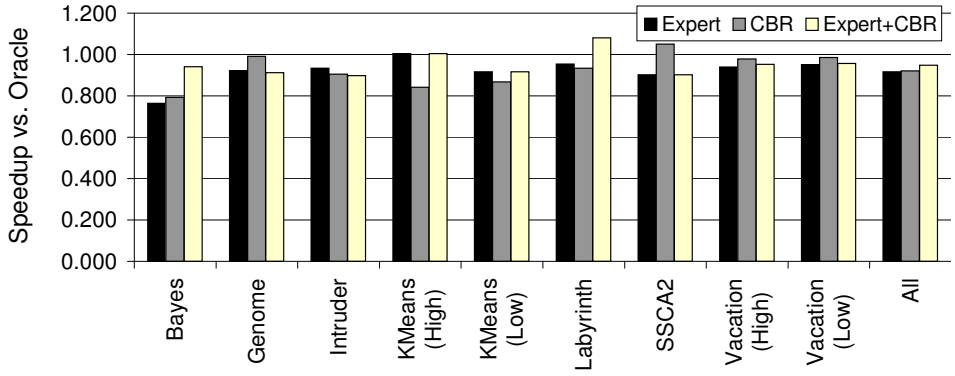

Fig. 5. Combining ExpertStatic with CBR D_TxTime+D_NonTxWork. For ELA semantics, the overall performance increases to 0.95. X semantics (not shown) reaches a 0.93 speedup.

rithm to select. When the most profitable cases are exhausted, the policy employs machine learning. Furthermore, the training set for the ML-based fallback excludes those cases handled by the expert. Figure 5 shows performance under ELA semantics when combining expert and ML policies. Overall performance increases to 0.95. With weak semantics, the policy achieves 0.93 speedup.

On a benchmark-by-benchmark basis, our hybrid policies always matched the best performance of the sub-policies on which they were based. Thus even though we have not yet evaluated ML-based policies trained on static profiles, we believe that this technique of combining expert intuition with an ML-based fallback is likely to provide the best performance. In particular, we believe the technique of letting the expert completely specify the use of high-risk, high-reward algorithms is a significant advancement over previous ML-influenced systems research.

8. FUTURE WORK

Section 7 focused on performance on single chip "Nehalem"-class x86 systems. To fully demonstrate the generality of a ML system for adaptive TM, further evaluation on other architectures (both other ISAs and other versions of the x86 ISA) is needed. We briefly summarize our findings for the STAMP Vacation benchmark on a 1.165 GHz, 64-way Sun UltraSPARC T2 with 32 GB of RAM, running Solaris 10. On this "Niagara2"-class machine, individual cores are very simple (in-order, single issue), but 8-way threaded (using fine-grained hardware multithreading). The L2 cache has low access times and is shared among all cores, but `CAS` instructions are slow as they are implemented out of core at the L2. These characteristics have a noticeable effect on Vacation.

— The "Mutex" STM implementation was the best performer at 1 and 2 threads. On the x86, Mutex was only best at 1 thread. This observation requires a redesign of "ThrX" and "ThrELA" policies.
— TLRW always outperforms TLRWLazy for low contention, and TLRWLazy always outperforms TLRW for high contention workloads. On the x86, TLRWLazy was always faster.
— The difference in performance between LSA and OrecLazy was much smaller.
— NOrec performed poorly on the Niagara (it was among the best "ELA" performers on x86).

To evaluate the implications of these observations, we created a few simple expert policies for the Niagara2. In each case, Mutex is used at 1 and 2 threads, and a single algorithm for all higher thread counts (we tested up to 56 threads, which is the limit for our version of TLRW). We call these policies NiagLSA, NiagOrecLazy, NiagTLRW, and NiagTLRWLazy. Their performance for X and ELA semantics is summarized in Table VIII.

In these specific cases, it is quite easy for the expert policy to perform extremely well. Nonetheless, it clearly holds that the expert policy must be architecture-aware. As future work, we intend to explore whether less dramatic variations in architecture (e.g., implementations of the same ISA)

Table VIII. Harmonic mean speedup (vs. Oracle) of STAMP Vacation on the Niagara2.

	Low		High			Low		High	
Algorithm	X	ELA	X	ELA	Algorithm	X	ELA	X	ELA
TLRWLazy	0.75	0.80	0.76	0.83	NiagTLRWLazy	0.87	0.93	0.88	0.97
TLRW	0.82	0.88	0.65	0.72	NiagTLRW	0.92	0.99	0.71	0.8
LSA	0.89	N/A	0.9	N/A	NiagLSA	0.98	N/A	0.99	N/A
OrecLazy	0.88	N/A	0.85	N/A	NiagOrecLazy	0.99	N/A	0.97	N/A

would result in new tradeoffs, and also the degree to which expert policies can perform well on the Niagara2 for more challenging workloads.

Another direction is the exploration of adaptive policies for Hybrid TM (HyTM) systems [Dalessandro et al. 2011; Riegel et al. 2011]. One of the biggest questions here relates to detecting when to adapt. In HyTM, aborts due to conflict are more common. Aborts also occur due to capacity limitations, transient faults (e.g., TLB misses), context switches, and forbidden instructions. The coarse mechanism of consecutive aborts may not apply here, and perhaps it will be necessary to *learn* how to respond to various abort types. The payoff could be significant, particularly when transactions frequently overflow the hardware capacity. In such a scenario, a variation of our mechanisms could improve performance by switching to a pure STM.

In both HyTM and STM, another exciting opportunity lies in the use of low-level performance counter data as features for training and adapting. Yen recently proposed a suite of simple TM-specific counters to aid in profiling [Yen 2009]. Both Yen's counters, and general hardware performance counters, will likely provide a wealth of information for characterizing workloads, though not all features will have predictive value. We believe that the relationship between low-level hardware events and high-level program behaviors will be easier to explore by using ML techniques such as those presented in this paper.

There is a question of whether STM algorithms should be "adaptivity-aware". For example, one could tune algorithms like Nano and TML, so that they dynamically detect when they are inappropriate choices and force an adaptation. This technique could also enable quicker detection of phase changes, especially in workloads like Genome.

Another important direction is to ensure the training data provides good coverage of the true feature space of TM applications. While we used a variety of microbenchmarks based on past publications, other options, such as EigenBench [Hong et al. 2010] may ultimately provide better coverage. There are certainly tradeoffs; for example, EigenBench does not have a means of distinguishing between "multiword CAS" and "read N write N" patterns, and its classification of accesses into three contention categories does not map cleanly to hierarchical data structures, particularly trees. The use of a small number of targeted microbenchmarks to capture specific behaviors and EigenBench for a resilient backup training data set may ultimately be the best choice.

Further afield, while we currently support choosing among variations of a single algorithm, there is likely to be significant benefit from tuning an STM algorithm using ML. For example, many algorithms have been proposed and evaluated on single-chip systems with small core counts. The danger is that certain parameters have been hard-coded for the development system, which could cause poor performance on next-generation systems with much higher core counts. We expect that using ML to tune STM internal parameters, such as backoff and granularity of conflict detection, will deliver significant performance improvements.

9. CONCLUSIONS

We believe that adaptive synchronization is necessary for high-performance shared memory programs. To that end, this paper introduces a system for combining static analysis, low-overhead dynamic profiling, and machine learning. It also presents a set of simple program characteristics that are suitable for making adaptivity decisions, and shows that our system can use these features to create TM libraries that automatically improve their performance. To the best of our knowledge, it is the first ML-based adaptivity system for synchronizing parallel programs.

Our best performance came from combining expert knowledge with machine learning. This approach allowed the expert to completely specify how high-risk, high-reward STM algorithms should be used. It also simplified the task of training ML-based adaptivity policies, by removing from the training set cases difficult for the ML algorithms, but easily handled by the expert.

Our experiences show the power that automatic ML-based adaptivity offers for solving hard systems problems. The combination of performance, maintainability, and flexibility in ML systems (which can even be retrained after deployment) make them an appealing approach for maximizing performance despite the complexity and heterogeneity intrinsic to parallel computing.

REFERENCES

AAMODT, A. AND PLAZA, E. 1994. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *Artificial Intelligence Communications 7,* 1, 39–59.

AGAKOV, F., BONILLA, E., CAVAZOS, J., FRANKE, B., FURSIN, G., O'BOYLE, M. F. P., THOMSON, J., TOUSSAINT, M., AND WILLIAMS, C. K. I. 2006. Using Machine Learning to Focus Iterative Optimization. In *Proc. of the Intl. Symp. on Code Generation and Optimization*. New York, NY.

ANJI HOME. 2005–2010. ANJI: Another NEAT Java Implementation. http://anji.sourceforge.net/.

ANSARI, M., KOTSELIDIS, C., JARVIS, K., LUJAN, M., KIRKHAM, C., AND WATSON, I. 2008. Lee-TM: A Non-trivial Benchmark for TM. In *Proc. of the Intl. Conf. on Algorithms and Architectures for Parallel Processing*.

CAVAZOS, J. AND O'BOYLE, M. 2005. Automatic Tuning of Inlining Heuristics. In *Proc. of the ACM/IEEE Conf. on Supercomputing*.

CAVAZOS, J. AND O'BOYLE, M. F. P. 2006. Method-specific Dynamic Compilation Using Logistic Regression. In *Proc. of the 21st ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*. Portland, OR.

COHEN, W. 1995. Fast Effective Rule Induction. In *Proc. of the 12th Intl. Conf. on Machine Learning*. Lake Tahoe, CA.

COOPER, K. D., GROSUL, A., HARVEY, T. J., REEVES, S., SUBRAMANIAN, D., TORCZON, L., AND WATERMAN, T. 2005. ACME: Adaptive Compilation Made Efficient. In *Proc. of the ACM Conf. on Languages, Compilers, and Tools for Embedded Systems*. Chicago, Illinois, USA.

DALESSANDRO, L., CAROUGE, F., WHITE, S., LEV, Y., MOIR, M., SCOTT, M., AND SPEAR, M. 2011. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory.

DALESSANDRO, L., SPEAR, M. F., AND SCOTT, M. L. 2010. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proc. of the 15th ACM Symp. on Principles and Practice of Parallel Programming*. Bangalore, India.

DICE, D., SHALEV, O., AND SHAVIT, N. 2006. Transactional Locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*. Stockholm, Sweden.

DICE, D. AND SHAVIT, N. 2010. TLRW: Return of the Read-Write Lock. In *Proc. of the 22nd ACM Symp. on Parallelism in Algorithms and Architectures*. Santorini, Greece.

FELBER, P., FETZER, C., AND RIEGEL, T. 2008. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*. Salt Lake City, UT.

FURSIN, G., MIRANDA, C., TEMAM, O., NAMOLARU, M., YOM-TOV, E., ZAKS, A., MENDELSON, B., BARNARD, P., ASHTON, E., COURTOIS, E., BODIN, F., BONILLA, E., THOMSON, J., LEATHER, H., WILLIAMS, C., AND O'BOYLE, M. 2008. MILEPOST GCC: Machine Learning Based Research Compiler. In *Proc. of the GCC Developers' Summit*. Ottawa, Canada.

GOTTSCHLICH, J., VACHHARAJANI, M., AND SIEK, J. 2010. An Efficient Software Transactional Memory Using Commit-Time Invalidation. In *Proc. of the 2010 Intl. Symp. on Code Generation and Optimization*. Toronto, ON, Canada.

HARRIS, T. AND FRASER, K. 2003. Language Support for Lightweight Transactions. In *Proc. of the 18th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*.

HARRIS, T., LARUS, J., AND RAJWAR, R. 2010. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool.

HARRIS, T., PLESKO, M., SHINAR, A., AND TARDITI, D. 2006. Optimizing Memory Transactions. In *Proc. of the 27th ACM Conf. on Programming Language Design and Implementation*. Ottawa, ON, Canada.

HONG, S., OGUNTEBI, T., CASPER, J., BRONSON, N., KOZYRAKIS, C., AND OLUKOTUN, K. 2010. Eigenbench: A Simple Exploration Tool for Orthogonal TM Characteristics. In *Proc. of the IEEE Intl. Symp. on Workload Characterization*. Atlanta, GA.

KULKARNI, M., BURTSCHER, M., INKULU, R., PINGALI, K., AND CASCAVAL, C. 2009. How Much Parallelism is There in Irregular Applications? In *Proc. of the 14th ACM PPoPP*. Raleigh, NC.

KULKARNI, P., HINES, S., HISER, J., WHALLEY, D., DAVIDSON, J., AND JONES, D. 2004. Fast Searches for Effective Optimization Phase Sequences. In *Proc. of the 25th ACM Conf. on Programming Language Design and Implementation*. Washington, DC.

LAU, J., PERELMAN, E., AND CALDER, B. 2006. Selecting Software Phase Markers with Code Structure Analysis. In *Proc. of the Intl. Symp. on Code Generation and Optimization*. New York, NY.

LEV, Y., MOIR, M., AND NUSSBAUM, D. 2007. PhTM: Phased Transactional Memory. In *Proc. of the 2nd ACM SIGPLAN Wkshp. on Transactional Computing*. Portland, OR.

LI, X., GARZARÁN, M. J., AND PADUA, D. 2004. A Dynamically Tuned Sorting Library. In *Proc. of the Intl. Symp. on Code Generation and Optimization*. Palo Alto, CA.

LI, X., GARZARÁN, M. J., AND PADUA, D. 2005. Optimizing Sorting with Genetic Algorithms. In *Proc. of the Intl. Symp. on Code Generation and Optimization*. Washington, DC, USA.

MARATHE, V. J., SCHERER III, W. N., AND SCOTT, M. L. 2005. Adaptive Software Transactional Memory. In *Proc. of the 19th Intl. Symp. on Distributed Computing*. Cracow, Poland.

MENON, V., BALENSIEFER, S., SHPEISMAN, T., ADL-TABATABAI, A.-R., HUDSON, R., SAHA, B., AND WELC, A. 2008. Practical Weak-Atomicity Semantics for Java STM. In *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*. Munich, Germany.

MINH, C. C., CHUNG, J., KOZYRAKIS, C., AND OLUKOTUN, K. 2008. STAMP: Stanford Transactional Applications for Multi-processing. In *Proc. of the IEEE Intl. Symp. on Workload Characterization*. Seattle, WA.

MONSIFROT, A., BODIN, F., AND QUINIOU, R. 2002. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *Proc. of the 10th Intl. Conf. on Artificial Intelligence: Methodology, Systems, and Applications*. Varna, Bulgaria.

NI, Y., WELC, A., ADL-TABATABAI, A.-R., BACH, M., BERKOWITS, S., COWNIE, J., GEVA, R., KOZHUKOW, S., NARAYANASWAMY, R., OLIVIER, J., PREIS, S., SAHA, B., TAL, A., AND TIAN, X. 2008. Design and Implementation of Transactional Constructs for C/C++. In *Proc. of the 23rd ACM Conf. on Object Oriented Programming, Systems, Languages, and Applications*. Nashville, TN, USA.

POUCHET, L.-N., BASTOUL, C., COHEN, A., AND CAVAZOS, J. 2008. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. In *Proc. of the 29th ACM Conf. on Programming Language Design and Implementation*. Tuscon, AZ.

RIEGEL, T., MARLIER, P., NOWACK, M., FELBER, P., AND FETZER, C. 2011. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proc. of the 23rd ACM Symp. on Parallelism in Algorithms and Architectures*.

SHEN, X., ZHONG, Y., AND DING, C. 2004. Locality Phase Prediction. In *Proc. of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. Boston, MA.

SONMEZ, N., HARRIS, T., CRISTAL, A., UNSAL, O. S., AND VALERO, M. 2009. Taking the Heat Off Transactions: Dynamic Selection of Pessimistic Concurrency Control. In *Proc. of the 23rd Intl. Parallel and Distributed Processing Symp.* Rome, Italy.

SPEAR, M. 2010. Lightweight, Robust Adaptivity for Software Transactional Memory. In *Proc. of the 22nd ACM Symp. on Parallelism in Algorithms and Architectures*. Santorini, Greece.

SPEAR, M. F., MICHAEL, M. M., SCOTT, M. L., AND WU, P. 2009. Reducing Memory Ordering Overheads in Software Transactional Memory. In *Proc. of the 2009 Intl. Symp. on Code Generation and Optimization*. Seattle, WA.

SPEAR, M. F., MICHAEL, M. M., AND VON PRAUN, C. 2008. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*. Munich, Germany.

STANLEY, K. AND MIIKKULAINEN, R. 2002. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation 10,* 2, 99–127.

STEPHENSON, M. AND AMARASINGHE, S. 2005. Predicting Unroll Factors Using Supervised Classification. In *Proc. of the Intl. Symp. on Code Generation and Optimization*. Washington, DC, USA.

STEPHENSON, M., AMARASINGHE, S., MARTIN, M., AND O'REILLY, U.-M. 2003. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proc. of the 24th ACM Conf. on Programming Language Design and Implementation*. San Diego, CA.

USUI, T., SMARAGDAKIS, Y., BEHRENDS, R., AND EVANS, J. 2009. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *Proc. of the 18th Intl. Conf. on Parallel Architecture and Compilation Techniques*. Raleigh, NC.

WELC, A., SAHA, B., AND ADL-TABATABAI, A.-R. 2008. Irrevocable Transactions and their Applications. In *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*. Munich, Germany.

YAO, X. 1999. Evolving artificial neural networks. *Proc. of the IEEE 87,* 9, 1423–1447.

YEN, L. 2009. Signatures in Transactional Memory Systems. Ph.D. thesis, University of Wisconsin, Madison.

YOTOV, K., LI, X., REN, G., CIBULSKIS, M., DEJONG, G., GARZARAN, M., PADUA, D., PINGALI, K., STODGHILL, P., AND WU, P. 2003. A Comparison of Empirical and Model-driven Optimization. In *Proc. of the 24th ACM Conf. on Programming Language Design and Implementation*. San Diego, CA.