Towards Applying Machine Learning to Adaptive Transactional Memory *

Qingping Wang†, Sameer Kulkarni‡, John Cavazos‡, Michael Spear†

†Lehigh University, ‡University of Delaware

{qiw209, spear}@cse.lehigh.edu, {skulkarn, cavazos}@cis.udel.edu

Abstract

There is tremendous diversity among the published algorithms for implementing Transactional Memory (TM). Each of these algorithms appears to be well suited to certain workloads and architectures. However, for programs that operate in distinct phases, exhibit input-dependent behavior, or must run on many different classes of machine, the best algorithm cannot be selected before the program actually runs. We introduce a mechanism for dynamic profiling of a running transactional program, and show how the profile can be used with machine learning techniques to select a TM implementation at run-time. Our preliminary results on the STAMP benchmark suite show good performance, providing a baseline for future research into adaptivity mechanisms for TM.

1. Introduction

Software Transactional Memory (STM) [15] algorithms differ in how they detect conflicts (using ownership records, signatures, or values), when they detect conflicts (on first write access to a location or at commit time), how speculative writes are performed (write-back with redo logs or write-through with undo logs), how conflicts between readers and writers are detected (using visible reads, invisible reads with validation, or invalidation), the granularity of conflict detection, what features are supported (retry and irrevocability), and what guarantees the STM makes to the programming language (especially semantics/privatization).

Each dimension has a cost that is difficult to quantify. Privatization seems to require either visible reads or program-wide synchronization on transaction commits. For workloads with high conflict rates, encounter-time locking with write-through can livelock; more expensive commit-time locking algorithms can be livelock-free. Workload characteristics including read-only ratio, frequency of irrevocable transactions, and nontransactional work between (and within) transactions affect which algorithm performs best. Furthermore, the cost of atomic operations and memory fences on a particular chip, the number of threads, and the cost of sharing data on multi-chip platforms all influence the choice of algorithm.

Even if a single algorithm could be "best" in all cases, performance could still benefit from specializing the algorithm at runtime, e.g., by adding a fast path for read-only transactions, optimizing backoff parameters and contention management, and implementing privatization barriers based on the number of chips and threads. In short, STM implementations will need to adapt if they are to deliver peak performance, and at least some of the adaptivity must depend on the environment of the running program.

* 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.

We introduce a predictive mechanism and dynamic profiling system to measure the behavior of a running program and change the STM algorithm to improve performance. We take as inspiration recent applications of machine learning (ML) to solve systems problems [7, 17, 29, 38]. Previous studies developed novel ML-based solutions for efficiently selecting compiler optimizations [2, 6, 7, 13], finding the best values for transformation parameters [5, 27, 37], and choosing the best algorithm to use for a particular sequential task [20, 21], to name a few examples. While ML-based solutions have shown much promise, they have not yet been used to improve the performance of parallel programs.

This paper introduces a ML system that leverages off-line learning to select an algorithm based on characteristics gathered by dynamic profiling. Our evaluation on the STAMP benchmark suite [4], shows good performance, though still 7% worse than a perfect "oracle" policy. In some cases our policies can exploit hard-to-find dynamic changes in program behavior (i.e., program phases [16, 18, 30]) to achieve higher performance than possible with any single algorithm.

After discussing previous adaptive STM systems in Section 2, we present our dynamic profiling framework in Section 3. We then describe our ML-based classifiers in Section 4, and discuss their use in adaptive STM. We present experimental results in Section 5, and then conclude with future research directions in Section 6.

2. Previous Adaptive STM Systems

Previous efforts to support adaptivity in STM either sought to prevent pathologies, or 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 [28, 40].

Location-Level Adaptivity Sonmez et al. dynamically change the concurrency control mechanism for *individual variables*, allowing those involved in frequent transaction conflicts to be accessed pessimistically [32]. This improves conflict detection and prevents some pathologies, without requiring pessimism on all accesses. However, supporting this mechanism requires some overhead on every access, to identify the variable's access mode.

Scalable Progress Guarantees Ni et al. proposed a "universal" algorithm [28] built atop a high-performance orec-based STM. They allowed concurrent "obstinate" transactions (using visible reads), as well as switches to serial and serial-irrevocable modes.

This system employed a novel indirection-based interface to prevent overhead while supporting these mechanisms, and through careful engineering, was able to also avoid global coordination when switching the mode of a transaction. That is, the instrumentation for any transaction seamlessly handled the fact of other transactions concurrently operating in other modes.

Performance Via Feature Monitoring ASTM [23] tracked when a workload employed a special API call ("early release", which removes a location from the transaction's read log), to indicate whether locations should be locked on first access or at commit time. This technique increased throughput and lowered latency for transactions, but was reliant on the use of an uncommon feature.

Re-Parameterizing the STM Felber et al. dynamically selected the number of orecs used for concurrency control [12]. This technique avoided false conflicts by increasing the number of available orecs, while workloads without concurrency could decrease latency by restricting themselves to a small set of orecs. An automatic mechanism, based on on-line analysis and simulated annealing, found the best number of locks for a workload.

Phased Execution PhTM [19] 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, except for avoiding pathologies. In addition, some variants required shared-memory communication at the beginning of some transactions even when there was no mode switch in progress, which could act as a bottleneck.

Selecting Locks or Transactions Usui et al. employed a combination of static and dynamic analysis to identify workloads for which locks outperformed STM, even when multiple threads were available [39]. 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, then at higher thread counts, the concurrency afforded by STM may not be worth its cost.

Pathology Avoidance The latest version of RSTM [33] supports adaptivity among different STM algorithms by combining the ideas from PhTM [19] with the indirection-based interface of Ni et al. [28]. The system selects from 10 algorithms, to react to bad performance. Decisions are based on an algorithm's likelihood of pathology and precision of conflict detection.

3. Dynamic Profiling

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 for I/O, condition synchronization, and early release, as well as the detection of pathologies (measured by consecutive aborts). While the first three of these characteristics are properties of a workload, the last is a characteristic of the combination of workload and STM algorithm.

In related work, Liu and Spear show that the abort rate of a workload, as well as the peak consecutive abort count of transactions in that workload, may have little relationship to throughput [22]. Observing a high abort rate when workload W runs under STM algorithm A does not indicate that a switch to algorithm A' is justified. Likewise, observing a low abort rate for $W + A$ does not mean that A' should *not* be chosen.

In order to devise a general framework that can accommodate a plurality of fundamentally different STM algorithms, we include dynamic characteristics that are not related to the choice of STM algorithm. Since API calls for I/O, condition synchronization, and early release may not be present in all workloads, we use the following features instead:

Shared Memory Accesses We count five types of shared-memory accesses that affect STM latency:

- ROReads: These are reads performed by transaction T *before its first write*. ROReads typically have the lowest latency (for example, in buffered-update STM systems, these reads do not require a write-set lookup).
- RAWReads: Reads to locations for which T has a speculative write often have low overhead.
- RWReads: With buffered update, reads performed by T after it has performed at least one write must include the cost of a write-set lookup that does not succeed.
- Writes: The number of distinct locations written by T.
- WAWWrites: A write to a location that has already been written may have lower costs.

Since few STM implementations optimize for read-after-read and write-after-read, we do not count these accesses separately.

Nontransactional Work When the gap between transactions is large relative to the duration of transactions, the best STM algorithm is typically one with low single-thread latency [39]. When the nontransactional work within a transaction is high (this could be due to accessing private or constant memory, or due to a high rate of arithmetic), the best STM algorithm is typically one with few implementation bottlenecks.

Writer Frequency For most STM algorithms, read-only transactions do not modify shared metadata. When the rate of writer transactions is low, these systems scale almost perfectly. As writers increase in frequency, the point at which read-only optimizations cease to be profitable varies with the STM algorithm.

3.1 Measurement Strategy

The above features all deal with the latency of individual transactions. As discussed in Section 4, we use off-line training to learn the relationship between these features and throughput. This simplifies our system considerably: we can turn off concurrency during profiling. This has the added effect of decreasing variance due to memory contention.

To measure shared memory accesses, we created a simple runtime, called ProfileTM. ProfileTM uses a fair ticket lock to prevent concurrent execution, and there is no global per-access metadata. Transactions buffer all writes until commit time, to ensure compatibility with code that uses self-abort and prevent some possible races between self-aborting transactions and nontransactional code [31]. Read and write instrumentation counts the five access types described above. ProfileTM also uses the hardware tick counter to estimate transaction running time. Switching to the ProfileTM algorithm and running N consecutive transactions generates an approximation of the behavior of transactions in the current phase of a running program. Since we use a ticket lock, these transactions are likely to represent a sample across many threads.

To estimate work between transactions and read-only frequency, we modified all of the STM algorithms in our system to count read-only and writing transactions separately. In addition, on every transaction commit we log the value of the tick counter, and on every transaction begin, we subtract that value from the tick counter, and add the result to a per-thread accumulator.[1] Dividing this value by the total number of transactions approximates the work between transactions. The nontransactional work within a transaction is derived by comparing the transaction's running time to a linear combination of counts of the 5 types of memory accesses.

[1] This count does not include time spent waiting to begin a transaction.

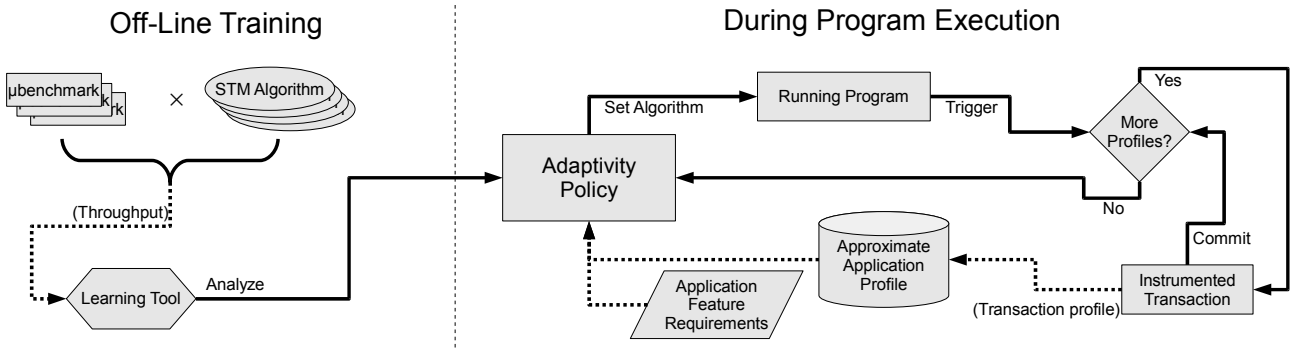

Figure 1. Adaptivity Workflow: A policy is trained for a machine by measuring the running time of parameterized microbenchmarks for all available STM algorithms at many thread levels. During program execution, a variety of events ("triggers") cause the STM library to profile a fixed number of transactions (using ProfileTM) and then use the policy to interpret the profile and select a new algorithm. Algorithm selections must also incorporate application requirements, such as language-level semantics.

The cost to collect a dynamic profile is low. Transactions run faster under ProfileTM than under traditional STM algorithms, since ProfileTM forbids concurrency and avoids global metadata accesses. Furthermore, the use of the tick counter makes timing very fast, at the cost of some accuracy (sources of error include thread migration and frequency scaling).

3.2 The Dynamic Profiling Process

Figure 1 presents the workflow of our adaptive system. We begin by testing all available STM algorithms on a set of microbenchmarks, and then building an adaptivity policy by training on that data (details appear in Section 4). This policy is linked to our STM library so that it can be invoked during a workload's execution. Note that ProfileTM is included in the set of training algorithms, so that we know the characteristics of transactions in each microbenchmark. We assume that by including the output of ProfileTM in the learning process, the training routine will learn concurrency characteristics of the STM algorithms.

When configuring the STM library during program initialization, the adaptivity policy selects an initial algorithm. This selection currently takes as input the required semantics of the application (for the time being, we only select between Encounter-Time Lock Atomicity (ELA) [26] and weak semantics), and whether the application uses self-abort. Extension to include condition synchronization, I/O, and other API calls is be straightforward.

During execution, three events ("triggers") cause a reevaluation of the current STM algorithm: when a thread is created, when any thread aborts more than 16 consecutive times, or when a thread blocks for more than 2048 cycles when trying to start a transaction,[2] we switch to ProfileTM. Whenever we switch to ProfileTM, in-flight transactions, including transactions waiting to acquire a lock, are not interrupted. Once these transactions complete, ProfileTM measures the next N consecutive transactions. Note that triggers occur off the critical path.

On every trigger, N transactions run, one at a time. Thus the sheer existence of our adaptivity mechanism ensures forward progress. 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 across all threads. On the next trigger, the same algorithm can be chosen only if the total commit count has increased. In this case the abort threshold for causing another trigger doubles.

4. Learning Algorithms

In the workflow and system described in Section 3, any classification algorithm can be used to guide ProfileTM's decision on which algorithm to select. We introduce two mechanisms, based on either case-based reasoning (CBR) [1] or neural networks using the neuro-evolution of augmenting topologies (NEAT) algorithm [36]. In Figure 1, the left hand side depicts how learning is performed: we measure microbenchmarks off-line, analyze them, and produce an executable policy. During execution, ProfileTM invokes this policy (after measuring N transactions) to select an algorithm.

4.1 Off-Line Training Data: Strategy

In a production environment, it is acceptable to tailor training data to the common-case for the target application. For the purposes of this paper, such an option is unfair. Instead, we train using parameterized microbenchmarks, and thus measure what should serve as a lower bound on the effectiveness of our adaptive system.

For each microbenchmark, we ran ProfileTM in single-threaded mode to describe the behavior (e.g., read-only ratio, transactional and nontransactional work, and the counts of the 5 types of memory accesses). We also measured throughput (averaged over five 5-second trials) at many thread levels using each of the STM algorithms supported by our library.[3] These two datasets were input to the learning tool. Since the collection of profiles at run-time is performed in a single-threaded environment, approximating a workload's behavior based on a single-threaded run is acceptable.

There are weaknesses in this approach. Our microbenchmarks (described below) are all homogeneous workloads with one program phase. We also assumed the per-feature average of roughly 15M profiled transactions for a workload is a fair approximation of the behavior of the workload. Furthermore, we considered feature magnitudes (e.g., number of reads from a read-only context), without quantization or normalization. For our proof-of-concept, these choices are acceptable, but we intend in future work to develop a more robust set of experiments and normalization strategies.

4.2 Off-Line Training Data: Workloads

In previous literature, differences in the configuration of microbenchmarks led to different STM algorithms offering maximum

[2] This applies to mutex-based STM systems and TML.

[3] For each application feature requirement scenario, only a subset of algorithms are considered.

throughput. By profiling these microbenchmarks to characterize their behavior, and then running them with every STM algorithm, at every thread level that does not cause preemption, we produced descriptions of how and when different characteristics may favor a particular algorithm on a particular architecture. We measured 103 benchmark configurations, using 15 different STM algorithms:

- Red-Black Trees: On our parameterized RBTree benchmark, we varied both the key range and the read-only ratio. Our trees hold 4-bit, 8-bit, 10-bit, 16-bit, or 20-bit keys, and have a mix of 0, 33, 50, 80, or 90% lookup transactions (the remaining transactions are equally split between inserts and removals). These experiments mirror workloads in which there is true concurrency, but real and false conflicts can occur.
- Linked Lists: We also considered linked lists storing 4-bit, 8-bit, and 10-bit keys, with the same 5 ratios of lookup/insert/remove transactions as in the RBTree experiments. These tests represent situations in which conflicts are frequent, but transactions have a large number of reads before any writes.
- HashTable: We included a hashtable microbenchmark with 8-bit keys, 256 buckets, and an equal lookup/insert/remove ratio. This workload has tiny transactions and few conflicts, and thus identifies bottlenecks in the STM algorithm. It also can indicate when the granularity of conflict detection is too coarse.
- Counter: In this workload, all transactions increment a single shared counter. This workload highlights the latency of transaction boundaries.
- WWPathology: This workload causes livelock under eager acquisition, and starvation in most other STMs [34].
- Disjoint: Transactions read and write to thread-private arrays. The workload should scale perfectly, with no aborts. When it does not, it indicates that conflict granularity is too coarse, that there are metadata bottlenecks in the STM implementation, or that polling for conflicts is too expensive or frequent. We parameterize so threads read either 10 or 100 locations per transaction, and write to 0, 10, 50, or 100% of the locations they read. There are also read-only transactions, which occur at a frequency of 0, 33, 50, 80, or 90%.
- Forest: A transaction accesses randomly-chosen RBTrees. The benchmark is parameterized by the number of trees in the forest (4 or 64), the key range of each tree (8-bit or 20-bit), the lookup likelihood of each tree operation (0% or 90%), and the number of tree operations per transaction (8 or 16). These workloads have unpredictable conflicts, but often with good scalability.

4.3 Case-Based Reasoning

The concept behind case-based reasoning (CBR) is that a system can learn from experience, and apply past experiences to make good decisions in the future [1]. During training, the system records every program behavior that it observed, the environment (e.g., thread count), and the best response (e.g., STM algorithm with the highest throughput). This collection of data is called the case base.

During program execution, our CBR policies scan the case base for entries that have the same number of threads as the workload. Among these entries, we select the one that is most similar to the average of the collected transactional profiles. The algorithm named by that entry, which corresponds to a peak performer for some microbenchmark configuration, is returned. We discuss candidate similarity metrics below.

CBR can support continuous learning, where decisions made during program execution are logged, along with the program behavior that led to that decision. We do not apply such techniques in our system, for two reasons. First, the quality of the descriptions of experiences is much more precise in our off-line training than it is when the case base is in use: this is because we never measure the full application with ProfileTM, but we do run each training microbenchmark with ProfileTM. Secondly, since we are trying to optimize a running program, we wish to avoid any overhead related to updating training data.

Similarity There are a total of 8 features that may be of use (average RORead, RWRead, RAWRead, Write, WAWWrite, and execution time; read-only ratio; and percentage of time spent of transactions). In our experiments, we evaluate each of these features in isolation. We also consider combinations of features. When comparing two rows in the case base, we use ratios instead of absolute numbers (e.g., the difference between 10 and 12 reads is the same as the difference between 100 and 120 cycles of execution time). We also normalize sums of features, so that a policy comparing reads (the sum of 3 distances) has the same weight as a policy comparing writes (the sum of 2 distances).

4.4 NEAT Classification

Neural network classifiers operate in a fundamentally different manner than CBR. For these systems, the 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 to implement a function that, for each vector I_k, computes the correct output o_k. The learned function is of very high dimension and complexity. 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 for any new input vector, will be able to output the best choice.

Among neural network classifiers, the most promising and powerful to date are based on augmented topologies [41]. These systems apply concepts of evolution to the creation of a network: a small graph serves as a starting point. Successive iterations of the unsupervised learning algorithm may add or remove nodes from the graph, may add or remove edges, and may change the weights of edges. Of those neural network algorithms that augment graphs, we use NEAT (Neuro Evolution of Augmenting Topologies) [36], which neither bounds the complexity of the output network, nor requires a complex starting point.

Unlike CBR, NEAT is a black box classifier. We cannot precisely explain why a particular input produces a particular output, nor can we explain which characteristics of an input favor a particular output. This property is both a strength and a weakness: while we cannot explain cases where NEAT fails to identify trends that we might think are obvious, NEAT has the potential to find relationships that are substantially more complex, nuanced, and robust than those that we specify for CBR.

5. Evaluation

We built our STM codes and adaptive policies on top of the adaptive version of RSTM [33]. The baseline adaptive RSTM provides 10 algorithms, and we added 5 more (marked with "*"). Unpublished algorithms are marked "(new)".

- Mutex: One test-and-test-and-set lock protects transactions.
- TML: A single-writer, multi-reader protocol [8].
- TMLLazy: TML with buffered update.
- OrecEager: Uses orecs, eager locking, and undo logs [12].
- OrecLazy: Uses orecs and commit-time locking [34].
- OrecFair: OrecLazy, with starvation avoidance and priority [34].
- OrecELA: Uses orecs but provides ELA semantics [25].
- NOrec: Uses values for conflict detection [9].
- NOrecPrio: Extends NOrec with a weak form of priority.
- RingSTM: Uses signatures for conflict detection [35].
- LLT*: The orec-based TL2 algorithm [10].
- ByteEager*: Uses byte locks for visible reads as in TLRW [11].

- BitLazy* (new): Similar to ByteEager, but with bitlocks [24], and commit-time locking.
- Inval*: InvalSTM [14] (signatures and visible reads).
- OptInval* (new): Inval, but with lock-free visible reads.

All experiments were performed on an HP z600 with 6GB RAM and a 2.66GHz Intel Xeon X5650 (Nehalem) processor with six cores (12 hardware threads). Code was compiled with g++ version 4.5.1, in 32-bit mode with –O3 optimizations. To evaluate NEAT-based prediction, we used the ANJI toolkit [3].

To train our adaptive policies, we tested all of the microbenchmarks from Section 4 using all of the above STM algorithms, at all thread levels that would not cause preemption. We input this data into our training algorithms to produce our adaptive policy components. We repeated this training, but without microbenchmark experiments for algorithms that offer weak semantics, in order to be able to assess the impact of language-level semantics on our adaptivity policies.

5.1 Baseline: Expert Adaptivity Policies

We measured the four policies that are included with RSTM [33], even though they are not expected to offer good performance, only pathology avoidance.[4] We also created several policies based on intuition about which algorithms are best at different thread levels. Two of these policies had strong performance: when no language-level semantics are required, ThrOrecEager uses Mutex at 1 thread and OrecEager otherwise. When language-level semantics (e.g., privatization safety, or "ELA" semantics) are required, ThrNOrec chooses Mutex at 1 thread, and NOrec at all other thread levels.

5.2 Evaluation Criteria

To evaluate our adaptive policies, we used the STAMP benchmark suite [4]. For each benchmark, we tested each of the 15 STM algorithms at 1, 2, 4, and 8 threads (these are the only supported thread levels that will not cause preemption on our 12-threaded system), and measured all of the recommended benchmark configurations.[5] All experiments are the average of 5 trials. Using this information, we created an "Oracle" dataset consisting of the best performer for each benchmark at each thread level.

Next, we ran ProfileTM on each of the STAMP benchmarks, and we collected the average counts for the 8 training features. For each of the CBR and NEAT adaptivity policies, we created a "predicted" dataset by determining what algorithm the policy would choose if given that whole-program profile, and selecting the corresponding running time. Finally, we ran each of the CBR and NEAT adaptivity policies on each benchmark, and measured the actual running time. To compare different policies, we summarize their performance using the harmonic mean speedup. Since this "average" speedup is over 36 benchmark/thread combinations, we also report the "best" and "worst" speedup observed across the 36 tests for each STM algorithm or adaptive policy.

Differences Between Measured and Predicted Performance We hope that a policy's predicted behavior is close to the oracle prediction, and its measured performance is close to its predicted behavior. Since the actual performance is based on adaptivity decisions driven by a small dynamic sample of the program's transactions, it is possible for it to differ in both directions from the predicted behavior. For example, if the dynamic sample differs from the averaged sample, then it could lead to selecting the same algorithm as the oracle predicts even though the average sample would predict a worse algorithm, or vice-versa.

[4] Only two of these policies support ELA semantics.

[5] We omitted the "yada" benchmark, since the released code intermittently crashes.

More interestingly, an adaptive policy can outperform the oracle. The oracle and the prediction assume that a workload's behavior is homogeneous. For workloads with multiple program phases [18, 30], or for which the amount of parallelism varies during execution [16], a policy might pick different algorithms at different points in the execution, and thus find a better overall strategy than can be achieved by predicting (even with perfect knowledge) a single algorithm for the whole workload. Note that it is not possible, in general, to statically detect the points at which algorithm re-selection should occur. Unpredictable dynamic properties, such as the length of the work queue or the number of completed mutations of a data structure, are often the best indicator of whether concurrent operations will conflict.

5.3 Early Results and Adjustments

On the STAMP labyrinth benchmark, our adaptivity policies performed poorly at 2 threads. They always picked a good algorithm, but they did not do so until the workload was more than 85% complete. Until that time, they used "Mutex", with thread one performing all of the work. This outcome was due to our implementation of Mutex with exponential backoff. While waiting "too long" when acquiring the lock is an effective adaptivity trigger, the bound was too large for this particular workload. Substituting a FCFS ticket lock led to adaptivity occurring much earlier.

Additionally, we found that the Inval and OptInval algorithms never performed as well as their peers on STAMP. In retrospect, this makes sense: Inval is designed to provide fairness on workloads where different threads execute different types of transactions, but in STAMP, all threads execute the same mix of transactions.

5.4 Adaptivity Configuration and Costs

Most of the transactions within each STAMP workload have similar behavior. As a result, running several transactions when collecting a profile had little to no impact on the performance of our systems. For STAMP, a profile of one transaction sufficed in most cases. Rather than consider different profile sizes for different workloads (which arguably might have helped Bayes, since it has 15 different transaction call sites), we opted to run all tests using a profile size of 1.

Given this profile size, the cost of collecting a profile was uniformly small. Since ProfileTM has minimal instrumentation, a profiled transaction runs 6% faster than NOrec, though 30% slower than Mutex. The main cost of collecting profiles comes from the global synchronization: when a profile is initiated, all in-flight transactions must finish first, then the profiled transaction runs, and then concurrency is turned back on. We counted the number of profiles collected for each workload. For all but Labyrinth, the total number of transactions was well over 1M, and the largest number of profiles gathered was 72. For Labyrinth, the largest number of profiles gathered was 3, with just over 500 transactions total. These numbers suggest that the cost of profiling is small. In additional tests, we confirmed this observation: inducing 100 profile collections in a workload with 1M transactions and 8 threads leads to negligible overhead.

5.5 Performance with No Semantics Requirements

Our first set of experiments explores the effectiveness of our adaptivity mechanisms in a setting where no language-level semantics are required, and all STM algorithms can be chosen. This scenario models a program phase where static analysis or programmer annotations ensure that no shared datum is accessed both with transactions and with nontransactional (uninstrumented) loads and stores. Figure 2 compares the best algorithms and adaptive policies in this setting.

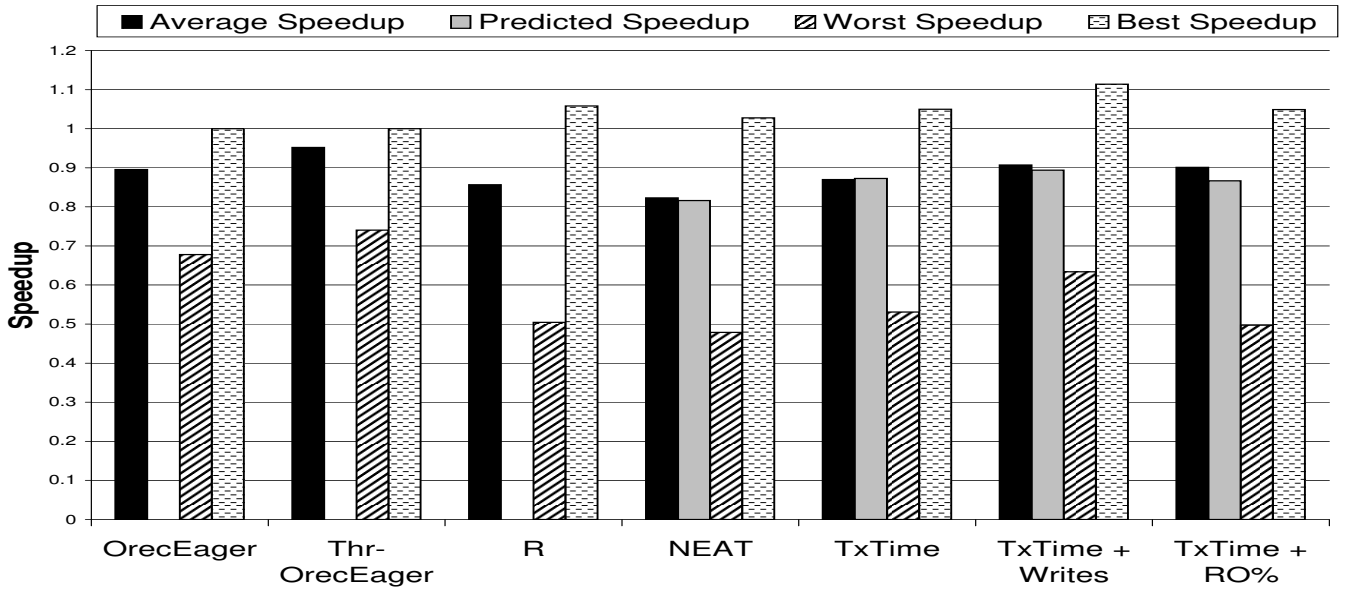

Figure 2. Performance of adaptivity policies on STAMP for 1/2/4/8 threads, using all 15 STM algorithms. The three rightmost columns are CBR policies, identified by the set of features they used.

The OrecEager algorithm (also known as LSA or TinySTM [12]) is best out of the 15 algorithms we evaluate. Its speedup, normalized to the oracle, is 0.895, with a worst speedup of 0.678. More impressively, at 4 and 8 threads, OrecEager's performance is above 0.95. Thus the ThrOrecEager policy (achieved by choosing the Mutex runtime at one thread, and OrecEager otherwise) achieves a harmonic mean speedup over oracle of 0.952.

The best prior adaptivity policy in this setting, Spear's "R" policy [33], only achieves a 0.85 speedup. Our NEAT policy fares worse, achieving only a 0.823 (we predicted it would achieve a 0.816). Our CBR policies fare better: the best single feature, transaction execution time (TxTime), achieves a 0.870 speedup; when we combine this with other features, we reach as high as 0.907.

We conclude that for the thread counts we were able to test, OrecEager is simply the best algorithm. Across 36 experiments, the OrecEager policy is best 18 times, and within 3% of the best choice in another 6 cases. It seems that our policies try too hard to find a complex policy, when a simple one is best. Whether this continues to hold at higher thread counts, or for workloads that are more diverse than STAMP, is left for future work.

5.6 Performance with ELA Semantics

When we require strong semantics from our STM algorithm, we lose the ability to choose OrecEager, OrecLazy, OrecFair, and LLT (TL2). We removed the data for these algorithms from the training dataset and trained new policies. Their performance, compared to an oracle that is given the same restricted set of algorithms, is reported in Figure 3.

In this setting, NOrec is the best performer, with a mean speedup of 0.861 compared to the oracle. The ThrNOrec policy increases performance to 0.904. The worst performance for both is 0.577 on KMeans (high contention) at 2 threads. NEAT delivers worse performance than ThrNOrec, scoring only a 0.823 (we predicted a performance of 0.813). NEAT's worst performance, also on KMeans (high contention) at 2 threads, was 0.505.

For CBR, again TxTime was the best single metric, achieving a speedup of 0.904 (we predicted 0.893), with a worst performance of 0.653 (on Bayes, at 8 threads). Performance on the troubling KMeans workload was 0.70. Furthermore, some combinations (always including TxTime) performed even better. By evaluating workloads using the transaction's running time, the average number of writes per transaction, and the percentage of time spent between transactions, the speedup compared to an oracle increased to 0.936.

While we used brute force across all combinations of our CBR metrics to come to this result, in retrospect it is quite obvious. To provide ELA semantics, many STM algorithms require blocking and polling to ensure a correct order between transactions and nontransactional accesses. Of the STM algorithms that block to provide ELA semantics, NOrec is usually fastest. Of those that do not require program-wide blocking to provide ELA semantics, ByteEager and BitLazy are fastest, but they have high single-thread latency. When writers are infrequent, transactions are long, or most threads are in nontransactional code, our best CBR policies choose NOrec. When these conditions do not hold, our policies are more likely to choose BitLazy and ByteEager.

5.7 Cost of Lookup

If a workload experiences frequent triggers, then it must frequently shut off concurrency and perform a global barrier to switch to the profiling mode. While this cost is unavoidable (and perhaps necessary for avoiding pathologies), an important related cost deals with how long the NEAT and CBR techniques take to compute the new algorithm. In our tests, NEAT decisions took 99K cycles on average, while CBR decisions averaged 150K cycles. NEAT variance was low, but CBR decisions could take up to 302K cycles.

Some of these differences in running time are due to implementation details, as NEAT processing uses floating point arithmetic, whereas we coded our CBR to use only integer math. Other differences relate to how training data is used during program execution. CBR systems must iterate through a large training set during execution to pick a new algorithm; the quality of predictions is typically proportional to the size of the set that is queried. NEAT, in contrast, processed a network with only 27 internal nodes and 108 connections. Typically, this is faster than scanning through more than a

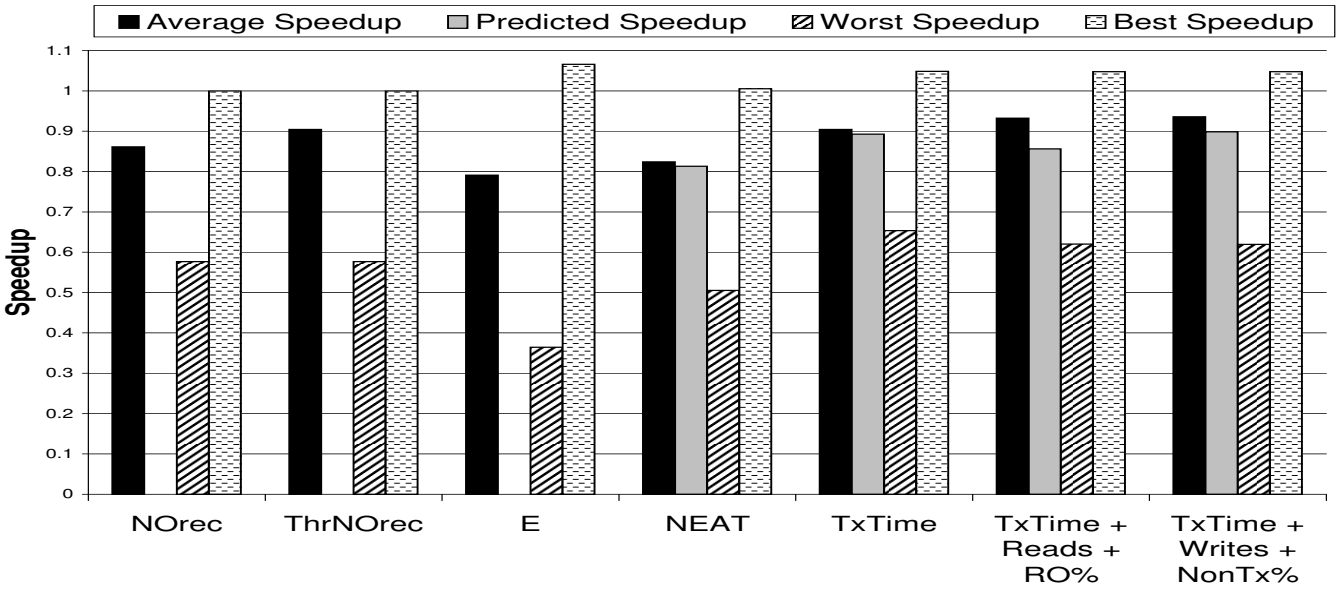

Figure 3. Performance of adaptivity policies on STAMP for 1/2/4/8 threads, using only STM algorithms with ELA semantics. The three rightmost columns are CBR policies, identified by the set of features they used.

hundred elements in a vector. It also is likely to cause fewer cache evictions.

5.8 Impact of Training Data

We selected our training data early in the process, and did not tailor it to our benchmarks. This led to an over emphasis on read-only transactions, even though many STAMP workloads consist exclusively of writing transactions. More significantly, we did not include training microbenchmarks where nontransactional work was a parameter. Consequently, for some STAMP benchmarks, there was no close training microbenchmark. When we added the STAMP benchmarks to our NEAT training data and performed leave-one-out cross validation, NEAT's performance rose by 4%.

5.9 Outperforming the Oracle

We were surprised to find several occasions where our adaptive policies outperformed the oracle by a large margin. The easiest benchmark for this outcome was Bayes, where the order in which transactions complete has a significant effect on total execution time. We saw a speedup of as much as 37% for some policies, which we attribute to those policies inducing a good order for threads to make progress.

In other cases, we observed consistent speedups in the range of 1% to 5%, compared to the oracle. We attribute this result to the program executing in phases, where each phase favors an STM algorithm with slightly different performance characteristics. This result was previously observed by Kulkarni et al. for worklist algorithms [16], where short periods at the beginning and end of the program have different characteristics than the middle, due to work becoming more, and then less, abundant. As STAMP contains many worklist algorithms, our results seem to support Kulkarni's observation.

6. Conclusions and Future Work

In this paper, we introduced a low-overhead system for dynamically profiling the behavior of memory transactions. We also proposed two adaptivity mechanisms based on machine learning that can exploit dynamic profiles to predict the STM algorithm that will maximize a workload's performance. By operating in this manner, our system is robust to program behaviors that are input-dependent, or that vary during distinct phases of execution.

We evaluated our system on the STAMP benchmark suite, considering both the case where STM algorithms must provide strong language-level semantics, and the case when relaxed semantics are acceptable. Our best policies performed within 10% of an oracle that always chooses the best algorithm. In the strong semantics case, our system was within 7% of the oracle.

Given that our system can exploit dynamic program phases to outperform the selection of a single STM algorithm, we can hardly claim that our current performance is at an acceptable level: for STAMP, it should be possible to outperform the oracle some of the time, and match its performance the rest of the time. To reach that end state will require better training benchmarks and more training features (especially nontransactional work).

In addition, we are not confident that our current strategy of learning the concurrency characteristics of STM algorithms from the training data is sufficient. We measured characteristics that affect latency, and expected the ML policies (especially NEAT) to infer properties related to throughput, such as metadata bottlenecks and conflict granularity. We were too optimistic. By incorporating (normalized) abort information directly into the training and profiling mechanisms, we hope to improve the accuracy of our prediction systems.

In the longer term, we believe that many more questions in TM research will be easier to address given our results, our mechanisms, and our framework (which will be released open-source). Opportunities include testing new ML algorithms, adapting in response to other STM feature requests (such as I/O), adapting on architectures for which hardware TM support is available, handling stronger language level semantics (especially for Java), and choosing among lock mechanisms for workloads that do not, or cannot, use transactions. Farther afield, we hope our profiling mechanism will simplify the task of building more general-purpose transactional profilers and debuggers.

References

[1] A. Aamodt and E. Plaza. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *Artificial Intelligence Communications*, 7(1):39–59, Mar. 1994.

[2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using Machine Learning to Focus Iterative Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, New York, NY, Mar. 2006.

[3] Anji Home. ANJI: Another NEAT Java Implementation, 2005–2010. http://anji.sourceforge.net/.

[4] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Seattle, WA, Sept. 2008.

[5] J. Cavazos and M. O'Boyle. Automatic Tuning of Inlining Heuristics. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, Nov. 2005.

[6] J. Cavazos and M. F. P. O'Boyle. Method-specific Dynamic Compilation Using Logistic Regression. In *Proceedings of the 21st ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, Oct. 2006.

[7] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. ACME: Adaptive Compilation Made Efficient. In *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, Chicago, Illinois, USA, June 2005.

[8] L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear. Transactional Mutex Locks. In *Proceedings of the Euro-Par 2010 Conference*, Ischia-Naples, Italy, Aug–Sep 2010.

[9] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 2010.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[11] D. Dice and N. Shavit. TLRW: Return of the Read-Write Lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[12] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[13] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, P. Barnard, E. Ashton, E. Courtois, F. Bodin, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O'Boyle. MILEPOST GCC: Machine Learning Based Research Compiler. In *Proceedings of the GCC Developers' Summit*, Ottawa, Canada, June 2008.

[14] J. Gottschlich, M. Vachharajani, and J. Siek. An Efficient Software Transactional Memory Using Commit-Time Invalidation. In *Proceedings of the 2010 International Symposium on Code Generation and Optimization*, Toronto, ON, Canada, Apr. 2010.

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

[16] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Cascaval. How Much Parallelism is There in Irregular Applications? In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

[17] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast Searches for Effective Optimization Phase Sequences. In *Proceedings of the 25th ACM Conference on Programming Language Design and Implementation*, Washington, DC, June 2004.

[18] J. Lau, E. Perelman, and B. Calder. Selecting Software Phase Markers with Code Structure Analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, New York, NY, Mar. 2006.

[19] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased Transactional Memory. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[20] X. Li, M. J. Garzarán, and D. Padua. A Dynamically Tuned Sorting Library. In *Proceedings of the International Symposium on Code Generation and Optimization*, Palo Alto, CA, Mar. 2004.

[21] X. Li, M. J. Garzarán, and D. Padua. Optimizing Sorting with Genetic Algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, Mar. 2005.

[22] Y. Liu and M. Spear. Toxic Transactions. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, San Jose, CA, June 2011.

[23] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sept. 2005.

[24] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, ON, Canada, June 2006.

[25] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, Sept. 2008.

[26] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[27] A. Monsifrot, F. Bodin, and R. Quiniou. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, Varna, Bulgaria, Sept. 2002.

[28] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Nashville, TN, USA, Oct. 2008.

[29] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. In *Proceedings of the 29th ACM Conference on Programming Language Design and Implementation*, Tuscon, AZ, June 2008.

[30] X. Shen, Y. Zhong, and C. Ding. Locality Phase Prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Oct. 2004.

[31] T. Shpeisman, A.-R. Adl-Tabatabai, R. Geva, Y. Ni, and A. Welc. Towards Transactional Memory Semantics for C++. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, Aug. 2009.

[32] N. Sonmez, T. Harris, A. Cristal, O. S. Unsal, and M. Valero. Taking the Heat Off Transactions: Dynamic Selection of Pessimistic Concurrency Control. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[33] M. Spear. Lightweight, Robust Adaptivity for Software Transactional Memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[34] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

[35] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the*

20th ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 2008.

[36] K. Stanley and R. Miikkulainen. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[37] M. Stephenson and S. Amarasinghe. Predicting Unroll Factors Using Supervised Classification. In *Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, Mar. 2005.

[38] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the 24th ACM Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[39] T. Usui, Y. Smaragdakis, R. Behrends, and J. Evans. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *Proceedings of the 18th International Conference on Parallel Architecture and Compilation Techniques*, Raleigh, NC, Sept. 2009.

[40] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[41] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, Sept. 1999.