EigenBench: A Simple Exploration Tool for Orthogonal TM Characteristics

Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun
Pervasive Parallelism Laboratory, Stanford University
{hongsup, tayo, jared, nbronson, kozyraki, kunle}@stanford.edu

***Abstract*—There are a significant number of Transactional Memory(TM) proposals, varying in almost all aspects of the design space. Although several transactional benchmarks have been suggested, a simple, yet thorough, evaluation framework is still needed to completely characterize a TM system and allow for comparison among the various proposals. Unfortunately, TM system evaluation is difficult because the application characteristics which affect performance are often difficult to isolate from each other. We propose a set of orthogonal application characteristics that form a basis for transactional behavior and are useful in fully understanding the performance of a TM system.**

In this paper, we present EigenBench, a lightweight yet powerful microbenchmark for fully evaluating a transactional memory system. We show that EigenBench is useful for thoroughly exploring the orthogonal space of TM application characteristics. Because of its flexibility, our microbenchmark is also capable of reproducing a representative set of TM performance pathologies. In this paper, we use Eigenbench to evaluate two well-known TM systems and provide significant insight about their strengths and weaknesses. We also demonstrate how EigenBench can be used to mimic the evaluation coverage of a popular TM benchmark suite called STAMP.

I. Introduction

Since the onset of transactional memory (TM) research, there has been an explosion in the number and variety of proposed TM systems. There are many design choices for TM, including how much functionality to put in hardware, types of conflict detection and resolution, choices for version management, overflow handling techniques, and many more [1]. The design space is now varied enough that evaluation of TM systems is an independent and interesting problem.

Attributes of a TM system affect the performance of a transactional application in different ways depending on the characteristics of the application. For example, applications with many shared transactional accesses may behave quite differently than applications with few. Application-based benchmarks are important since their patterns attempt to represent realistic workloads, but they have a limited ability to isolate the effect of each characteristic on the overall performance. For example, an application's working-set size is often tied to the size of its transactions, but these two characteristics may be completely orthogonal in terms of how they affect system performance. Current evaluation frameworks do not fully account for this reality.

In our approach, we first propose a set of orthogonal characteristics of TM applications, which we call *eigen-characteristics*. We show how together these characteristics form a basis for all TM applications in the same way that a basis in linear algebra spans a vector space. Ideal TM evaluation requires the ability to decouple the eigen-characteristics from each other and vary them independently, a methodology which we call *orthogonal analysis*. To this end, we have developed *EigenBench*, a simple yet thorough microbenchmark for fully evaluating a transactional memory system (Section II). EigenBench provides insight into a TM system's performance by fully exploring the eigen-characteristics, evaluating corners of the application space not easily reached by existing benchmarks (Section III).

In addition to application characteristics, it is useful to understand an implementation's susceptibility to *pathologies*, as they have been shown to significantly affect performance [2]. We show that EigenBench can easily reproduce a representative set of pathological cases (Section IV).

Finally, we show that the performance of TM applications can be easily explained in terms of their eigen-characteristics. We demonstrate how given these characteristics, EigenBench can closely approximate the execution behavior of that application for the purposes of evaluating a TM system. We then perform examples of this mimicry using benchmarks from the STAMP TM suite [3] (Section V).

A. Related Work

Other researchers have released TM microbenchmarks and benchmark suites. TM microbenchmarks are usually structured as a parameterizable set of transactions that operate on a shared data structure like a red-black tree or a hash table [4]–[8]. These tools are small, portable, and useful. However, they do not aim to parameterize across the entire TM design space and have not been shown to reproduce all applicable, known pathologies.

The STAMP benchmark suite [3] consists of eight configurable applications and exercises a wide range of transactional behaviors. Although STAMP successfully provides a representative set of TM applications, each application's transactional characteristics are strongly tied to its parameters and are thus difficult to decouple from each other. Being a full benchmark suite, STAMP is also more complex and requires more configuration before being used. In addition, STAMP has not been shown to exercise all pathological cases. We will demonstrate that EigenBench is flexible enough to mimic the transactional characteristics of the STAMP benchmark suite.

Other TM benchmark suites have been proposed, including those from Kestor et al. [9], Ansari et al. [10], and Guerraoui et al. [11]. These suites focus on particular application domains and are not designed to exercise the full breadth of TM behavior. Our approach is most like STMBench7 [11] in that we provide a single parameterized microbenchmark that can be easily configured to mimic a wide variety of workloads. However, our approach aims to be much more general and cover a larger portion of the TM application space by avoiding any particular underlying data structure. As an example, SwissTM outperformed TL2 by 3x in all STMBench7 results but only by 1.1 – 1.9x in STAMP results [12], which shows that STMBench7 did not produce performance behaviors of STAMP applications

Other researchers have also defined a set of characteristics to analyze and reproduce behaviors of existing benchmarks [13], [14]. However, their selection of characteristics were not orthogonal; they instead relied on principle component analysis (PCA) for their choices. Thus, orthogonal analysis such as what we present in Section III is not possible with those characteristics.

The specific contributions of this paper are:

- We define a set of orthogonal characteristics of TM applications, and show that an orthogonal analysis along these characteristics is extremely useful in understanding a TM system's behavior.
- We present EigenBench, a lightweight yet thorough microbenchmark designed for such an orthogonal analysis.
- We show that one can explain a TM application's performance given its eigen-characteristics and demonstrate how EigenBench can be used to approximate the behavior of real applications.
- We show that EigenBench can easily reproduce all of the pathological cases delineated by Bobba et al. [2] for the evaluated TMs.

II. EigenBench's Orthogonal Characteristics

In this section, we first define the eigen-characteristics: the set of orthogonal attributes that characterize TM applications. We then present EigenBench, a microbenchmark designed to orthogonally explore these attributes. Finally, we describe how to systematically vary those characteristics using the EigenBench parameters.

A. Eigen-Characteristics

It is well understood that applications having different characteristics vary in behavior within a single TM system [3], [13], [15]. However, there is no consensus on a standard set of characteristics by which to describe TM applications. Previous proposals have used sets of characteristics that are far from orthogonal; one may be strongly determined by others. For example, a set composed of *transaction size*, *read set size*, and *write set size* do not form an orthogonal basis, since the first depends on the others.

We propose *eigen-characteristics*, a set of orthogonal characteristics that can form a basis for all TM applications. Table I presents the eight characteristics: *concurrency, working-set size, transaction length, pollution, temporal locality, contention, predominance*, and *density*. While there can be alternative selections of orthogonal characteristics, our experience found this specific choice was intuitive and very useful in practice. By our definition, *predominance* presents the ratio of shared reads and writes to the entire application including all non-shared instructions, while *density* dictates, as a complementary measure, what fraction of the non-shared instructions are executed inside transactions.

A conventional (non-orthogonal) characteristic can now be expressed as a combination of eigen-characteristics. For example, read set size is a function of *transaction length*, *pollution*, and *temporal locality*. A small read set size can be obtained by having a small number of shared accesses, a small portion of reads among many shared accesses, or many repeated accesses to a few addresses. TM systems might vary substantially in how they handle these three cases.

Also, we are deliberate in our choice of adjectives used to describe the characteristics in Table I. We especially avoided the terms 'big' and 'small' since they are too vague: a 'big' transaction could mean a *long* one (having many shared reads and writes), a *long but non-repetitive* one (having large read/write set), or even a *sparse* one (having many instructions, either shared accesses or not, inside the TX).

Section III will show in detail how these characteristics are helpful in understanding the performance of a TM system.

B. EigenBench

Pseudocode for the main functions in EigenBench is displayed in Figure 1. At its core, the benchmark is fairly simple; each thread performs a pre-defined number of transactions then exits. A transaction consists of a set number of transactional reads and writes (Lines 12 - 18), with some non-transactional "local" operations interspersed (Line 20). Additional local operations are performed between each transaction (Line 24). Parameters used in the code, most of which are arguments to the `test_core` function, are described in Table II. Note that throughout the code we occasionally modify the dummy variable `val` simply to prevent the compiler from optimizing code away.

Three separate arrays are accessed in the code. `Array1` is the *hot* array, meaning it is shared between all threads. `Array2` is the *mild* array, which is also accessed transactionally. Each thread accesses its own partition of `Array2`, however, so accesses will not cause conflicts. `Array3`, the *cold* array, is partitioned like `Array2` but is used for non-transactional accesses. As we discuss in the next section, using three distinct arrays allows us to control the contention between the transactions and the amount of influence the non-transactional code has on the caching behavior of the workload.

The `rand_actions` function decides if the transaction should perform a read or a write and whether to access the hot or mild array. This function guarantees that the precise number of reads and writes to each array, as given parameter values, are eventually performed but the order in which they are

TABLE I
ORTHOGONAL TM CHARACTERISTICS

Characteristic	Definition	Descriptive Adjectives
Concurrency	Number of concurrently running threads	high-concurrent/low-concurrent
Working-set size	Size of frequently used memory	wide/narrow
Transaction length	Number of shared accesses per TX*	long/short
Pollution	Fraction of shared writes to shared accesses	dirty/clean
Temporal locality	Probability of repeated address per shared access	repeating/non-repeating
Contention	Probability of conflict of a transaction	contentious/low-conflicting
Predominance	Fraction of shared access cycles to total execution cycles	significant/insignificant
Density‡	Fraction of non-shared cycles executed outside transactions to total non-shared cycles†	dense/sparse

* *Shared accesses* means reads/writes that should be protected by TM. *Shared cycles* is execution cycles consumed by such accesses.
† *Non-shared cycles* is execution cycles consumed by any other instruction than shared accesses.
‡ We define density as 1.0 if there is no non-shared instructions in the application, which does not happen in practice.

```
1  void test_core(tid, loops, pesist, lct, R1, W1, R2, W2
2      R3_i, W3_i, Nop_i, k_i, R3_o, W3_o, Nop_o, k_o) {
3    long val=0;
4    long total = W1 + W2 + R1 + R2;
5    for (i=0; i<loops; i++) {
6      Save_Random_Seed;
7      BEGIN_TM();
8      if (persist) Restore_Random_Seed;
9      (r1,r2,w1,w2) = (R1,R2,W1,W2);
10     Reset_History_Buffers;
11
12     for (j=0; j<total ; j++) {
13       (action, array) = rand_action(r1, w2, r2, w2);
14       index = rand_index(tid, lct, array);
15       if (action == READ)
16           val += TM_READ(array[index]);
17       else
18           TM_WRITE(array[index], val);
19       if ((j%k_i)==0)
20          val += local_ops(R3_i, W3_i, Nop_i, val, tid);
21     }
22     END_TM();
23     if ((i%k_o)==0)
24       val += local_ops(R3_o, W3_o, Nop_o, val, tid);
25 } }
26 static long A1, A2, A3, N;
27 static long *Array1, *Array2, *Array3;
28 void init_arrays() {
29   Array1 = malloc(A1 * sizeof(long));
30   Array2 = malloc(A2 * N * sizeof(long));
31   Array3 = malloc(A3 * N * sizeof(long));
32 }
33 (Action, Array) rand_action(r1, w1, r2, w2) {
34  // With uniform random probability based on r1,w1,r2,w2
35  // randomly choose one among: {(Read Array1),
36  // (Write Array1), (Read Array2), {Write Array2}}
37  // And decrease corresponding variable by one.
38 }
39 long rand_index(tid, lct, array) {
40  // With probability of lct, choose a saved index
41  // from the history buffer of array, or
42  // randomly choose an index from range
43  // (0~A1) or (tid*A2 ~ (tid+1)*A2)
44  // and save it to the history buffer of array
45 }
46 long local_ops(R3, W3, NOP, val, tid) {
47  // Perform R3 reads and W3 writes
48  // on Array3[tid*A3 ~ (tid+1)*A3] in random order.
49  // Then perform NOP number of nops.
50 }
```

Fig. 1. PseudoCode description of EigenBench.

performed is randomized. `local_ops` performs a given number of read/write operations on the cold array, then performs *nop*s. The call to this function on line 20 splits up the transactional accesses within a transaction, and the call on line 24 splits up the transactions themselves. Note that `k_i` and `k_o` are scalers; local operations can be either more (`k=1`) or less (`k>1`) numerous than shared operations.

The address accessed is expressed as an index to the selected array, which is determined by the function `rand_index`. Depending on the `lct` parameter which determines temporal locality, this will be either a random index in the range or an index previously used and saved in the history buffer.

Note that we abstracted out all instructions other than transactional reads and writes but captured their effects (e.g. instruction mix, ILP, or branches) simply with the parameters α and *nop*s. We did this because our goal was to qualify the performance characteristics of TM systems specifically, not those of general systems. We thus provide the key knobs that directly affect the TM systems. We remind the reader that to a TM system, a user application is fundamentally just a series of random reads and writes in concurrent transactions; EigenBench abstracts the application as such.

C. Deriving Eigen-characteristics from EigenBench

In this section, we explain how EigenBench can be used to generate a specific application execution pattern with the desired transactional characteristics.

Table III summarizes how the eigen-characteristics can be derived from EigenBench parameters[1], with the exception of *contention*. For *contention*, we use an approximate expected value, which we compute as follows. Let us denote the number of unique addresses in hot array accesses as W_1' and R_1'. [2] We start with the estimate that a given access will cause a conflict with probability $(N-1)*(W_1')/A_1$, or the number of writes performed by all other transactions combined divided by the size of the shared array. [3] The complement of that event is an access not causing a conflict, which must happen $W_1'+R_1'$ times, giving us the probability of no conflicts. Finally, we take

[1]We defined working-set size based on per-thread memory usage. An alternative is to use total aggregated memory size. In that case, working-set size is derived as $A_1 + A_2 * N + A_3 * N$.

[2]W_1' is defined as follows: If $lct = 1$ then W_1' is 1. Else W_1' is $\lceil W_1 * (1 - lct) \rceil$. R_1' is defined similarly.

[3]This expression approximates the probability and is only valid when $A_1 >> W_1'$. For a better approximation, you can use $\hat{W}'$, the expected number of addresses occupied by N-1 other threads. This value can be obtained by the solution of the coupon collector's problem [16].

TABLE II
PARAMETERS USED IN EIGENBENCH

Name	Meaning	Name	Meaning	Name	Meaning
N	Number of Threads	R_1	Reads/tx of *Hot* array	W_3o	Writes of *Cold* array btwn TXs
S	Random Seed	W_1	Writes/tx of *Hot* array	Nop_i	No-ops between TM accesses
tid	Thread id	R_2	Reads/tx of *Mild* array	Nop_o	No-ops outside TX
loops	Number of TX per thread	W_2	Writes/tx of *Mild* array	K_i	Scaler for in-TX local ops
A_1	Size of Array1 (*Hot* array)	R_3i	Reads of *Cold* array inside TX	K_o	Scaler for out-TX local ops
A_2	Size of Array2 (*Mild* array)	W_3i	Writes of *Cold* array inside TX	persist	Restore random seed if violated
A_3	Size of Array3 (*Cold* array)	R_3o	Reads of *Cold* array btwn TXs	lct	Probability of address repetition

TABLE III
EIGENBENCH PARAMETERS USED TO DERIVE SPECIFIC CHARACTERISTICS

Characteristic	**Eigenbench Parameters**	**Characteristic**	**Eigenbench Parameters**
Concurrency	N	Working-set size	$A_1 + A_2 + A3$
Transaction length	$R_1 + R_2 + W_1 + W_2 =(T_{len})$	Pollution	$(W_1 + W_2)/T_{len}$
Temporal locality	lct	Contention	*see Equation (1)*
Predominance	$T_{len} * \alpha/(T_{len} * \alpha + C_{in} + C_{out})$	Density	$C_{out}/(C_{in} + C_{out})$
Read set Size*	$(R_1 + R_2) * (1 - lct)$	Write set Size*	$(W_1 + W_2) * (1 - lct)$

$N_{in} = ((R_{3i} + W_{3i}) * \alpha + Nop_i) * T_{len}/K_i, \quad O_{in} = \beta * T_{len} * (1 + (R_{3i} + W_{3i})/K_i), \quad C_{in} = N_{in} + O_{in}$
$N_{out} = ((R_{3o} + W_{3o}) * \alpha + Nop_o)/K_o, \quad O_{out} = \beta * (R_{3o} + W_{3o})/K_o, \quad C_{out} = N_{out} + O_{out}$
α: the average memory access latency β: overhead of random address generation and action decision in CPU cycles†

* We also include derivations for two important non-orthogonal characteristics: read set size and write set size.
† For simplicity of explanation, we assume (α, β) = (1,0) in the remaining of the paper.

the complement of that to get the probability of a conflict as shown in Equation 1. We remind the reader that the equation below estimates the degree of conflict induced by EigenBench with the given parameter values and not by any other general application. The validity of this equation will be demonstrated in Section III.

$$P_{conf} = 1 - \left(1 - \min\left\{1, \frac{(N-1)W_1'(1-lct)}{A_1}\right\}\right)^{W_1'+R_1'} \tag{1}$$

With EigenBench, it is always possible to adjust only one eigen-characteristic value while keeping the others fixed since there is always at least one free parameter for each attribute. For example, one can increase *transaction length* or *pollution* without increasing *contention* by controlling R_2 and W_2 only.

EigenBench is not limited to deploy uniformly characterized transactions across all threads. Rather, one can mix and match various characteristics in a single execution. For example, one thread can execute `test_core` function with one set of parameter values (e.g. long and dirty) while other threads execute with a different set of values (e.g. short and clean). Each thread can also execute different parameter sets over time, depending on modeled program 'phases'. Furthermore, one can slightly extend the code in Figure 1 such that the active parameters (e.g. R_1, W_1) become random variables themselves.

Finally, EigenBench can be easily extended to address other TM aspects such as nesting, strong atomicity and object-based TM. To address (the performance aspects of enforcing) strong atomicity, one can add accesses of shared arrays (Array1 and Array2) outside the transactions. A nested transaction can be implemented as recursive callings of test_core(). Object-based TM can also be addressed by replacing the long-typed arrays with an array of objects.

III. ORTHOGONAL ANALYSIS CASE STUDY: TL2 AND SWISSTM

In this section, we demonstrate how to perform orthogonal analysis using EigenBench to thoroughly analyze a TM system. Our evaluation features two high-performing STMs: TL2-x86 [5] (version 0.9.6 provided with STAMP [17] using the default GV4 versioned locks) and SwissTM [12] (version 2009-09-10 [18]). All systems are compiled with the `-m64 -O3` options. Our experiments were run on an HP ProLiant DL140 with two quad-core 2.33GHz Intel Xeon E5345 processors and 32GB of RAM. The E5345 has a total of 8MB of shared L2 cache. Throughout this section, we will italicize the names of eigen-characteristics as found in Table I.

Since our analysis involves a high-dimensional search space, we only vary one dimension (i.e. characteristic) at a time while fixing the others to their *typical* values. Table IV displays our choices for typical values. We are interested in medium length (*transaction length* of 100) and relatively clean (10% *pollution*) transactions. We also use independent transactions (zero *contention*) when inspecting the overhead of the TM system; we will analyze the effect of *contention* separately. Finally, we fix *density* and *predominance* to be their maximum values (1.0) in order to focus on TM overhead only. Of course, one may select other typical values and perform similar analyses according to his interests (e.g., focusing on very short transactions with *transaction length* < 10). We remind the readers EigenBench allows for variation of any characteristic

TABLE IV
DEFAULT CHARACTERISTICS IN FIGURE 2

Characteristics	**Value**	**Characteristics**	**Value**	**Characteristics**	**Value**	**Characteristics**	**Value**
Concurrency	8	Working-set size	256 (KB/thread)	Transaction length	100	Pollution	0.10
Temporal locality	0.0	Contention	0.00	Predominance	1.00	Density	1.00

TABLE V
VARIED EIGENBENCH PARAMETERS IN FIGURE 2.

Graph	N	A1	A2	A3	R1	W1	R2	W2	R3i	W3i	R3o	W3o	lct	Range
(default)	8	0	32k	0	0	0	90	10	0	0	0	0	0	
(a)	-	-	Var	-	-	-	-	-	-	-	-	-	-	A_2=1k ... 32m
(b)	-	-	-	-	-	-	Var	Var	-	-	-	-	-	$W_2 + R_2$=10 ... 520
(c)	-	-	-	-	-	-	-	Var	-	-	-	-	-	W_2=0 ... 100
(d)	-	-	-	-	-	-	-	-	-	-	-	-	Var	lct=0.125 ... 1.0
(e),(f)	-	Var	Var	-	45	5	45	5	-	-	-	-	-	$A_1 + A_2$=32k, A_1=1k ... 24k
(g)	-	-	16k	16k	-	-	-	-	-	-	Var	-	-	R_{3o}=1 ... 100
(h)	-	512	31k	512	8	2	82	8	Var	-	Var	-	-	$N_{in} + N_{out}$=100, R_{3o}= 1 ... 90
(i)	Var	-	-	-	-	-	-	-	-	-	-	-	-	N=1... 8

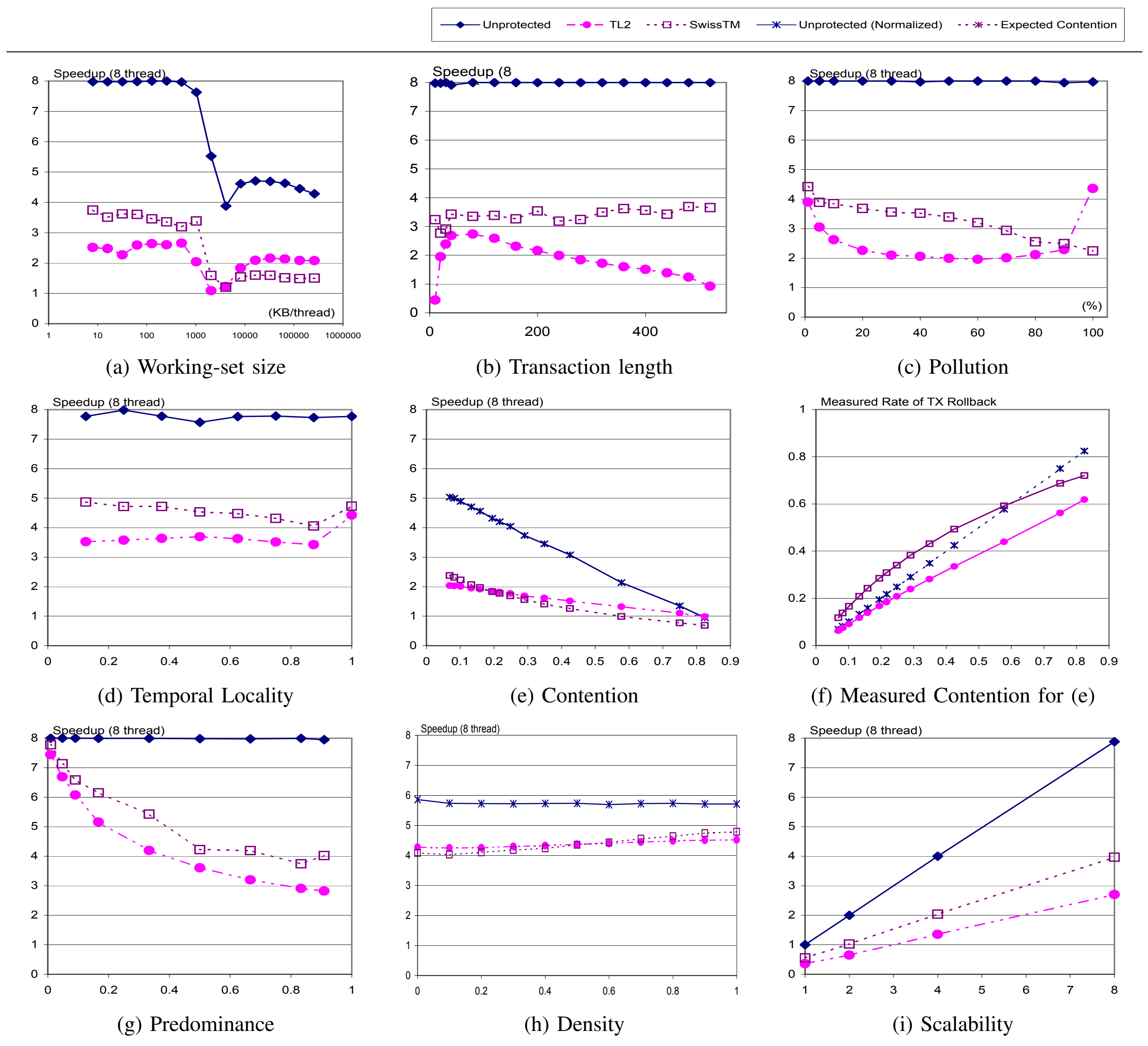


Fig. 2. Orthogonal analysis of two TM systems using EigenBench. For (h), (0.24, 0.83) were used for Contention and Predominance, respectively.

while keeping others fixed since there is always one or more free parameters to accomplish this.

Figure 2 depicts the results of the orthogonal analysis. The x-axis denotes the value of the characteristic in question; Table V describes the values used in each graph. The y-axis denotes the speedup results with eight threads (higher is better). We plot three lines: TL2, SwissTM, and *Unprotected* speedup which represents the case of a multi-threaded execution without the protection of a TM system or even locks (i.e. TM reads/writes are mapped to direct loads/stores and TM begin/end to null statements). The unprotected execution thus serves as an upper bound of available performance, even though the bound is loose since no atomic execution is guaranteed. Note that EigenBench trivially allows the unprotected execution since it never suffers crashes or infinite loops due to a breach of atomicity; the same is not true for some benchmarks due to their dependence on complex data structures (e.g. rb-tree insertion). Also, note that when *contention* is zero, the unprotected execution then represents a hard upper bound.

Graph (a) shows the effect of different *working-set sizes*. The dramatic drop for all systems at 2MB/thread is an effect of the cache hierarchy; four threads execute concurrently on a quad-core chip with a shared 8MB L2 size of 8MB. This is seen even in the unprotected version because the performance is bounded by off-chip memory access. However, we also observe other interesting performance patterns for TL2 and SwissTM: When the working set fits on-chip, SwissTM performs much better than TL2 but performs worse when the working set outgrows the caches. This phenomenon seems to be caused by SwissTM's larger locktable size.

Graph (b) explores the effect of *transaction length*. The first four x-values shown are 10, 20, 30, and 40 with all subsequent values spaced evenly by 40. We see immediately that TL2 is extremely susceptible to transaction length; it performs poorly for very small transactions (10) and also degrades quickly again after a certain threshold value ($\tilde{1}20$). On the other hand, SwissTM handles all lengths equally well. The original SwissTM paper [12] explains how it handles different transaction lengths with different strategies.

Graph (c) shows the effect of *pollution*, or fraction of transactional writes. As we increase the number of writes, TL2's performance first drops quickly and then flattens; SwissTM's drops much more slowly and thus performs better overall due to its better write buffer structure.

Graph (d) shows the effect of *temporal locality*, or fraction of re-used addresses in a transaction. The graph shows that SwissTM's performance drops somewhat as we increase locality, indicating that the system is optimized for unique addresses. Both systems perform well when only a single address is used in a transaction (lct=1.0).

Previous discussion has focused on TM system overhead in the absence of *contention*. We now examine behavior when this is not the case. Before discussing the results, we note two things. First, the horizontal axis is the value of *expected contention*, as calculated by Equation 1. Second, since the unprotected execution does not stall or rollback and thus results in an overly loose bound, we normalize by dividing the measured execution time by one minus the expected amount of contention. This normalized execution time is still not a true bound, but a more reasonable approximation.

Graph (e) shows the effect of *contention*. One may notice that all speedups, including unprotected, are now far below 8, even when expected contention is very low. This is an effect of the system's ccNUMA architecture. Since the *hot* array (size of A_1 « 8MB) is shared by eight cores on two CPUs, about half of the accesses to that array will be served from the other CPU, even when the access may not result in a conflict for the current transaction.

The difference in measured speedup between TL2 and SwissTM in graph (e) is also interesting. We see that SwissTM performs only slightly better than TL2 when there are few conflicts and becomes worse with increasing degrees of contention. This is the result of two factors. First, SwissTM is more sensitive to the off-chip accesses induced by the cc-NUMA architecture, as we saw in graph (a). Second, SwissTM rolls back more transactions than TL2, as will be seen in graph (f).

Graph (f) depicts the rate of transaction rollbacks as we vary the degree of expected contention. The graph shows that SwissTM has a higher rollback rate than both TL2 and the expected number of conflicts for the regular access patterns of the benchmark. This is because SwissTM detects conflict with quadword granularity while TL2 does so with word granularity and thus SwissTM exhibits false positives. The case of more irregular accesses will be discussed in Section V.

The results in graph (e) and (f) shows that SwissTM's performance is comparable to or better than TL2, even when it has twice high rollback rates. This suggests to TM designers that it makes sense to trade-off more false positives in conflict detection for less barrier overhead, since TM mostly targets low-conflicting applications.

Note that graph (f) serves to validate Equation (1). The graph clearly shows the high correlation between estimated degree of conflict (dotted line) and two actual violation rates reported by two STMs (solid lines); A TM design may allow a higher violation rate (e.g. false-positive filters) or a lower rate (e.g. transactional locks) than the 'real' conflicts.

Graph (g) shows the effect of *predominance*. As we decrease the *predominance* factor, the performance of both systems approaches that of the unprotected execution. Thus if shared accesses are rare, it makes no difference which system is used. As *predominance* increases, however, the plot indicates a measure of overall TM system overhead. SwissTM introduces less overhead than TL2 for the accompanying set of typical values.

Graph (h) depicts the effect of *density*. Note that we changed two of the typical values for this measurement. First, we set *predominance* to be 0.5. That is, in the entire program, there are as many non-TM instructions as there are TM instructions. In changing *density*, we change the proportion of non-TM instructions located outside transactions, with 1 being the maximum. Second, we choose a non-zero value for expected *contention*, 0.19, because it increases the re-execution penalty for adding more instructions inside transactions.

In graph (h), as we decrease *density* (i.e. putting more instructions into transactions), the performance drops slowly due to increased rollback penalty, but not much. The re-execution is cheaper than the first execution since much of fetched data still remain in its cache. Thus, a TM programmer may consider merging two short transactions interleaved by a short non-transactional section of code, since very small transactions may pose larger overhead in some TM systems (See graph (b)).

Graph (i) exhibits the scalability of two TM systems. Both systems scaled well with the non-conflicting transactions while SwissTM still maintained much less overhead as in previous graphs.

We now summarize our findings from the orthogonal analysis case study. We showed that TL2's performance drops quickly with long or dirty transactions (graphs (b) and (c)) while SwissTM is neutral to these characteristics. We also showed that SwissTM performs much better than TL2 only when the working-set size fits in the cache, but can get worse otherwise (graph (a)). Also SwissTM is shown to rollback more transactions than TL2 for regular data access pattern, mostly due to false positives in conflict detection. The results also indicate that it is reasonable to trade off barrier overhead and conflict detection accuracy (graphs (e) and (f)). Note that benchmark studies in the original SwissTM paper [12] did not reveal all these aspects but merely stated SwissTM performed better than TL2 in all cases.

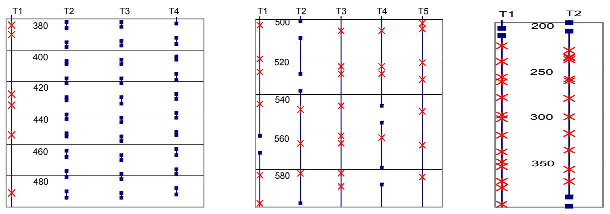
We omit further in-depth analysis on why one STM performs better than the other in specific cases, since this requires a thorough understanding of the details of the STM implementations and is out of scope of this paper. A TM system designer, however, would be able to use this analysis to properly evaluate trade-offs and gain more insight. One may even continue to analyze additional design spaces using different sets of typical values, or a mixture of transactions with different characteristics. Examples of such further exploration can be found in the Appendix.

IV. Pathology Generation

In the previous section, we showed how EigenBench can be used to analyze TM systems using orthogonal analysis. EigenBench can also be used to generate particular workloads which exercise TM systems in specific ways. In this section, we generate workloads that generate pathological TM behavior.

Transactional memory performance pathologies degrade performance by unnecessarily hindering the progress of transactions. Bobba et al. [2] showed that hardware TM performance notably improves by addressing the potential for pathologies in the system. Refer to the original paper [2] for detailed descriptions of all the pathologies.

Based on Bobba et. al.'s description of the pathologies, we built a set of EigenBench parameters that can generate



(a) StarvingElder (b) RestartConvoy (c) FriendlyFire

Fig. 3. Diagram of Observed Pathological Executions: The y-axis shows timestamps. Solid lines indicate the duration of a transaction; the dots represent begin/end for a transaction; and crosses are the re-start of a transaction.

each pathological behavior, depending on the underlying TM system implementation. The set of parameters is found in Table VI. Note that our choice of parameter values were arbitrary as long as they matched the description of the pathology situation in the paper [2]; any similar values produced the same pathology.

We present verification of the EigenBench parameters for three of the pathologies: StarvingElder, RestartConvoy and FriendlyFire. Our verification platform is the default TL2-x86, which uses lazy conflict detection, and its eager variant. Both versions use lazy version management and are provided with the STAMP benchmark suite, as in the previous section. For verification, we first add a simple global time stamp to the system; a dedicated thread increases this global variable on a regular basis. When a thread begins or ends a transaction, the event is stored in a local event history buffer with the current time stamp value. We analyze the data off-line to create a graphical representation of the execution sequence, as shown in Figure 3.

The pathology signatures are evident in the figure. In (a) StarvingElder, a long transaction (T1) cannot make progress due to shorter transactions (T2 $\sim$ T4) constantly violating it. In (b) RestartConvoy, a commit (e.g. T4 after 540) rolls back many other transactions and they all restart at about the same time. In (c) FriendlyFire, two threads (T1 and T2) violate each other continuously.

We omit verifications for other pathologies; SerializedCommit applies only to HTMs and all the others only to TM systems that use eager conflict detection and eager version management. If provided these systems, however, one could easily utilize EigenBench with its simplicity and flexibility to generate the remaining pathologies.

V. Characterization of Actual Workloads

In previous sections, we have shown that by varying the eigen-characteristics using EigenBench, we can effectively analyze a TM system. We will now demonstrate that there is an actual mapping from a real application to a set of characteristics. Of course, real applications usually have a mix of various transaction types and exhibit complex memory access patterns. In this section, we address these while answering the following three questions:

TABLE VI
EIGENBENCH PARAMETERS FOR GENERATION OF PATHOLOGICAL CASES

Pathology	**EigenBench Parameters**													
(TM Design Point)	N	A1	A2	A3	R1	W1	R2	W2	R3i	W3i	R3o	W3o	lct	persist*
FriendlyFire (EL)	2	1k	16k	8k	100	20	400	0	200	0	0	0	0	1
StarvingElder (LL)	1	1k	-	8k	128	32	0	0	0	0	100	100	0	0
	7	1k	-	8k	2	2	0	0	0	0	100	100	0	0
RestartConvoy (LL)	8	8	4k	8k	4	2	20	20	5	0	100	100	0	1
SerializedCommit(LL)	8	-	16k	8k	0	0	10	500	0	0	0	0	0	0
StarvingWriter (EE)	1	32	-	1m	0	30	0	0	500	0	0	0	0	0
	7	32	-	-	30	0	0	0	0	0	0	0	0	0
FutileStall (EE)	8	1k	16k	-	80	20	10	10	0	0	0	0	0	0
DuelingUpgrade (EE)	2	128	-	2m	80	10	0	0	20	0	0	0	0.75	0

* The *persist* option compels the transaction to repeat the same address sequence if it retries (see Fig. 1). This causes some pathologies occur more frequently.

- Can we get (or approximate) eigen-characteristics for real applications?
- If so, what characteristic (or sets of characteristics) dictates the performance of those applications?
- Can Eigenbench reproduce the performance behavior of those applications, given their eigen-characteristics?

In this analysis, we use applications from the STAMP benchmark suite [3]. We are first able to identify some of the eigen-characteristics through profiling: *transaction length, working-set size, pollution, and locality*.When measuring *locality*, we use a unit of four words. STMs based on manual instrumentation provide an easy way to achieve this profiling: We simply log shared accesses in single-threaded execution and perform off-line analysis. Appropriate values for *predominance* and *density* are estimated from application documentation or using source code analysis, although exact values could be obtained via instrumentation-based profiling or simulation. Recall from Section III that *predominance* is simply a scaling factor for all TM systems, and performance sensitivity to *density* is small. Finally, since an exact value for *contention* is hard to capture, we use the rollback percentage, reported by the TM system, as a good approximation. Table VII summarizes the results.

Please note that certain applications exhibit large variations in characteristic values; some could be described as having several modes of operation. For example, the first column in Figure 4 shows a histogram of transaction lengths for the five STAMP applications featured. The transaction lengths of Genome and Intruder have long "tails" in the graph, while Labyrinth is rather uniformly distributed. On the other hand, Vacation-Low has a normal distribution and SSCA2 is single-valued. In the second column of Figure 4, we provide diagrams of the access frequency per address region; this illustrates the memory access pattern and working-set size of each application.

Our next question is determining what characteristics dictate the performance behaviors of these applications. We answer this question while reproducing these behaviors using EigenBench. In this study, we manually obtained EigenBench parameters that capture the collected eigen-characteristics in Table VIII, since they are discrete abstractions of the continuous characteristics shown in the first two columns of Figure 4. Note that a single parameter set was enough to abstract some applications, while multiple sets were needed for others, depending on the variance of the characteristics.

We present the EigenBench result in the third column of Figure 4; the graphs display the speedup and transaction rollback rates for each application. We plot four lines for each graph: two solid lines from the original application executed with TL2 and SwissTM as well as two dotted lines from the EigenBench executions with the parameters in Table VIII.

For **Genome**, the governing characteristic was *transaction length* or, more precisely, a mixture of different lengths. In Genome, the dominant transaction length is relatively small (< 100); however, it has a long tail, meaning larger transaction lengths are used with nontrivial frequency. We capture this dynamic property by using three different parameter sets (Table VII); this was essential to capture the performance behavior. SwissTM's better scalability with 8 threads is a result of its sophisticated conflict resolution mechanism which prevents overly frequent aborting of long transactions. Also note that, due to this mechanism, SwissTM now exhibits a lower rollback rate than TL2, unlike the analysis in Section III.

Vacation-Low exhibits very different key characteristics. *Transaction length* is somewhat normally distributed, but the application uses a much wider range of memory. As a result, the limitation becomes cache miss latency resulting from large *working-set sizes*. We captured this with a single set of parameter values as shown in Table VII.

Labyrinth has a uniform distribution over *transaction length* and a narrow memory footprint, as in the case of Genome. Although the transaction size is relatively long for this application, it has an extremely low *density* (meaning many non-TM instructions inside transactions) and low *predominance* (many non-TM operations overall). This suggests that it is not the overhead of transactional read/write operations, but rather the efficiency of conflict detection that governs its performance.

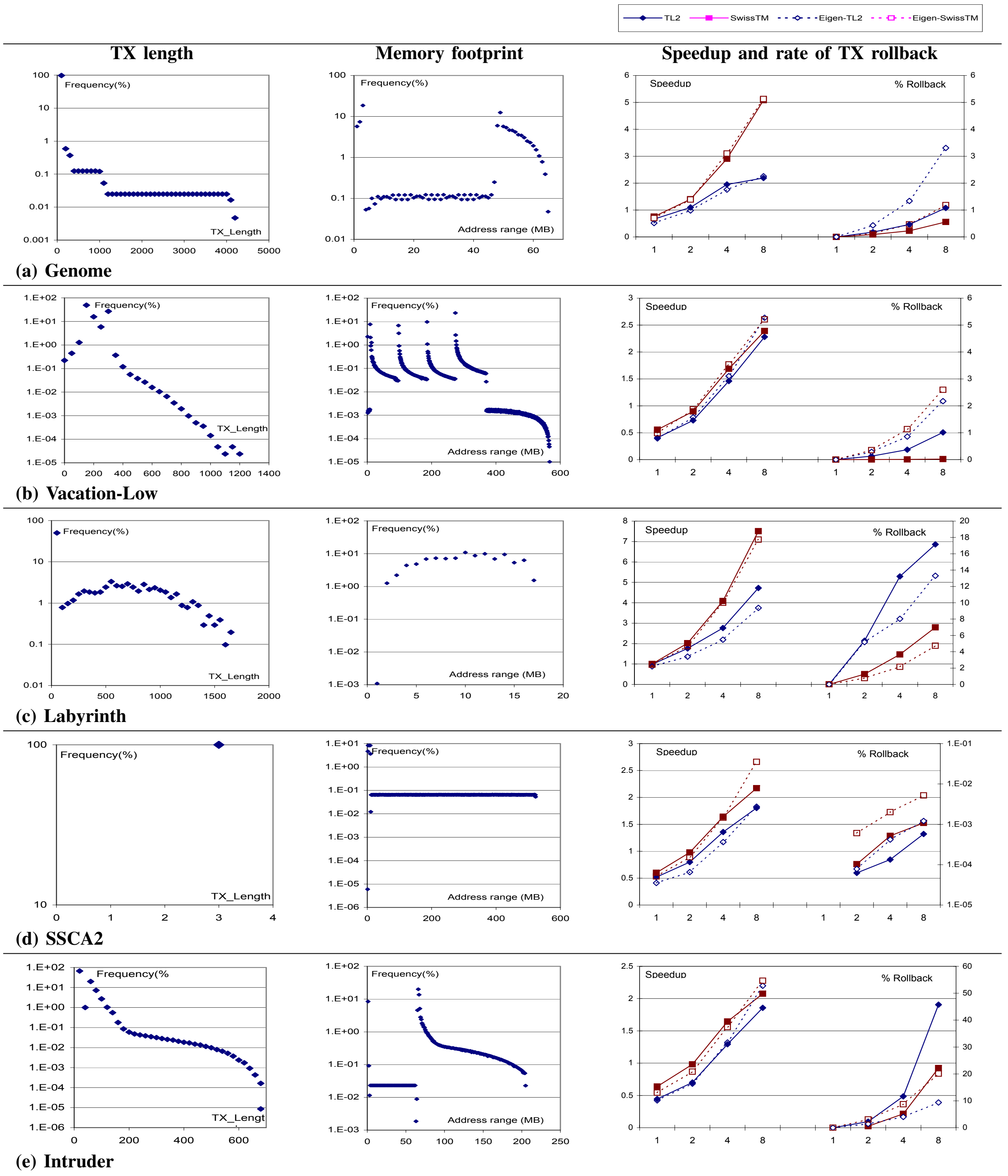


Fig. 4. Characteristics and Performance of STAMP Applications and EigenBench: The first two columns are histograms; each presents an Eigen-characteristic (TX length and Working-set size) for the application in each row. The last column compares the measured performance and rollback ratio of the original STAMP application (solid lines) and EigenBench executions with parameters set as in Table VIII (dotted lines). Note that the Eigen-characteristic values for EigenBench execution can be deterministically calculated from those parameters.

TABLE VII
EIGEN-CHARACTERISTICS OF STAMP APPLICATIONS

Application[†]	**Working-set**	**TX Length**	**Pollution**	**Locality**	**Contention**	**Predominance**	**Density**[*]
Genome	20 MB	88, (1..4000)	5%	0.58	0.5%	High	High
Vacation-low	256 MB	226, (1.. 1239)	2%	0.59	0.2%	High	High
Vacation-high	256 MB	180, (1.. 1878)	4%	0.55	0.4%	High	High
Labyrinth	16 MB	357, (3..1688)	50%	0.77	5%	Low	Low
SSCA2	400 MB	{ 3 }	33%	0.33	0.0005%	Low	High
Intruder	20 MB	24, (3..680)	5%	0.52	22%	Low	High
Kmeans-low	8MB (8KB)	{2, 66}	50%	0.81	25%	Low	High
Kmeans-high	8MB (8KB)	{2, 66}	50%	0.81	43%	Low	High
Yada	200MB	45 (1..2194)	28%	0.47	22%	High	High

[*] For working-set of Kmeans, the number inside parenthesis is the amount of memory addresses accessed inside TX. For TX length, we present average value and range. For pollution and locality, we present measured average if the distribution is unimodal, or top n-mode values if multi-modal. Contention is estimated by the minimum value of the percentage of transactions that roll back on the given TM systems with 8 threads. We loosely estimated predominance and density from source code analysis. Also see the first two columns of Figure 4 which display variance of tx-length and memory footprint.

[†] Bayes failed to execute in our 64-bit environment and is excluded from this study.

TABLE VIII
EIGENBENCH PARAMETERS FOR MIMICKING SELECTED STAMP APPLICATIONS.

App.	A1	A2*N	A3	mix(%)	R1	W1	R2	W2	R3o	W3o	R3i	W3i	LCT
Genome	256k	2m	256k	97%	28	2	65	5	20	20	0	0	0.5
				2%	145	5	340	10	80	80	0	0	0.5
				1%	770	30	1660	40	500	500	0	0	0.5
Vacation-low	512k	32k	64k	100%	198	2	47	3	0	0	0	0	0.6
Labyrinth	128k	2m	64k	55%	10	5	0	0	5	5	0	0	0.7
				15%	25	25	50	50	0	0	600	600	0.7
				15%	25	25	275	275	0	0	600	600	0.7
				15%	25	25	650	650	0	0	600	600	0.7
SSCA2	32m	8k	64k	100%	1	1	1	0	15	0	0	0	0
Intruder	1k	1k	64k	70%	8	2	0	0	2	2	0	0	0.5
				28%	18	2	27	3	4	4	0	0	0.5
				2%	20	10	90	10	10	10	0	0	0.5

SSCA2 shows a very uniform data access pattern as illustrated in the second column of Figure 4 and Table VII. In this case, short *transaction lengths* and large *working-set size* govern the performance.

Intruder displays a variety of *transaction lengths* and a wide range of addresses accessed. It also displays a large amount of *contention*, as shown in Table VII. The performance is mainly limited by the contention and short transactions.

Overall, EigenBench was able to closely approximate TM application behavior by reproducing key eigen-characteristics of the applications. Please note that all of these applications exhibit complex (pointer-chasing) memory access patterns, but the approximations were still reasonably close. We remind the reader that the goal of our analysis is not to provide 100% identical replay of the target application; we are interested in capturing a few key characteristics and simulating those characteristics with EigenBench.

We omit graphs of other applications' behavior but discuss their key characteristics here. **Vacation-High** has a large *working-set size* similar to **Vacation-Low**, and this governs its behavior. However, it exhibits more *contention* than Vacation-Low. Both versions of Kmeans have very narrow *working-set sizes*, short transactions and high *locality*; these further emphasize the effect of barrier overhead. Like Genome, **Yada** shows large variance in *transaction length*, but it has a much wider memory footprint. Also, transactions are more contentious and more dirty than in Genome.

To summarize, we have shown that one can obtain (mixed sets of) eigen-characteristics for real applications, but there are only a few key characteristics that dictate the performance behavior of an application. Along with results from an orthogonal analysis (as performed in Section III), knowledge about the characteristics of TM applications can explain the performance behavior of such applications as exemplified in this section. A TM developer can use this information to further improve a system. For example, to improve the performance of the Vacation application, one must minimize the TM system's impact on cache pressure while efficiently handling long transactions.

We have also shown that using EigenBench, one can reasonably model certain application behaviors given (sets of) eigen-characteristics. This is clearly evident in the similarities between the EigenBench and TL2/SwissTM executions, displayed in the third column of Figure 4. Conversely, the performance behavior of EigenBench with a certain set of parameters accurately represents the performance behavior of complex applications which exhibit those characteristics. In other words, since the eigen-characteristics successfully explain performance behavior of a TM application, a through exploration using EigenBench allows analysis of a TM system across an extremely wide range of applications not covered by other benchmarks.

VI. Conclusion

Previous transactional memory benchmark suites and microbenchmarks have often lacked the ability to isolate the key attributes of an application and determine how each affects the performance of the TM system. Additionally, there is little consensus in the literature on which application characteristics accurately and independently represent realistic application behavior. In this paper, we proposed eigen-characteristics, a set of orthogonal application characteristics which are very useful in analyzing a TM system. Together, the eigen-characteristics capture the dominant behavior of TM applications. To provide a simple way to explore these attributes, we presented EigenBench, a lightweight yet powerful parameterizable microbenchmark. As a demonstration, we used EigenBench to analyze two prominent STM systems. EigenBench allowed us to vary each characteristic independently, providing insight on how they affected the systems in the absence of other interfering factors. In addition, we showed that the eigen-characteristics can successfully specify real TM applications by using EigenBench to accurately reproduce the behavior of several STAMP applications. In doing so, we demonstrated a concrete mapping between real applications and (sets of) eigen-characteristics. We also showed how EigenBench can be used to generate important specific execution patterns by reproducing three TM pathological cases.

Acknowledgements

This work is supported by DOE contract, Sandia order 942017; Army contract AHPCRC W911NF-07-2-0027-1; DARPA contract, Oracle order 630003198; and Stanford PPL affiliates program, Pervasive Parallelism Lab.

References

[1] J. Larus and R. Rajwar, *Transactional Memory*. Morgan Claypool Synthesis Series, 2006.

[2] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," in *ISCA '07*.

[3] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC '08*, September 2008.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *ASPLOS'06*.

[5] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*. Springer, March 2006, pp. 194–208.

[6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *ASPLOS '09*.

[7] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory," in *PPoPP '06*.

[8] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg, "McRT–STM: A high performance software transactional memory system for a multi-core runtime," in *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[9] G. Kestor, S. Stipic, O. Unsal, A. Cristal, and M. Valero, "Rms-tm: A transactional memory benchmark for recognition, mining and synthesis applications," in *TRANSACT'09: 4th workshop on transactional computing*.

[10] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis, "Lee-tm: A non-trivial benchmark suite for transactional memory," *Lecture Notes in Computer Science*, vol. 5022, pp. 196–207, 2008.

[11] R. Guerraoui, M. Kapalka, and J. Vitek, "STMBench7: A Benchmark for Software Transactional Memory," in *Proceedings of the Second European Systems Conference (EuroSys2007)*, 2007.

[12] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," in *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2009, pp. 155–165.

[13] C. Hughes, J. Poe, A. Qouneh, and T. Li, "On the (dis)similarity of transactional memory workloads," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 108–117.

[14] J. Poe, C. Hughes, and T. Li, "TransPlant: A Parameterized Methodology For Generating Transactional Memory Workloads," in *MASCOTS'09: IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*.

[15] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun, "Characterization of TCC on Chip-Multiprocessors," in *PACT '05*.

[16] I. Adler, S. Oren, and S. Ross, "The coupon-collector's problem revisited," *Journal of Applied Probability*, vol. 40, no. 2, pp. 513–518, 2003.

[17] "STAMP: Stanford transactional applications for multi-processing," http://stamp.stanford.edu.

[18] "SwissTM," http://lpd.epfl.ch/site/research/tmeval#swisstm.

[19] "EigenBench," http://ppl.stanford.edu/eigenbench.

Appendix

In this section, we provide a short introduction to using EigenBench for analysis of a TM system. Eigenbench is publicly available from our website [19]. We provide source code that can be easily executed in conjunction with any STM or HTM implementation or simulation. The package also includes specific EigenBench parameters discussed in the paper.

(1) Orthogonal analysis: The main usage of EigenBench should be the orthogonal overhead analysis as in Section III. First, one may specify a typical transaction description of interest (e.g. transaction length and pollution), making sure to use non-conflicting characteristics. At a minimum, one should explore working-set size, transaction length, pollution and scalability, and compare the results against *unprotected* execution (see Section III). This will reveal the true overhead induced by the TM system. Results for locality, predominance and density may be omitted in the report if they don't provide further insights beyond those reported in this paper. Finally, performance under contention and measured rollback rates should be explored.

(2) Mixed transactions: Optionally, one may want to analyze performance under non-uniform transactional characteristics (e.g. long transactions mixed with short transactions). For this purpose, we provide multiple sets of parameters in our distribution package, based on our application analysis. Given very mixed parameters, it is not easy to obtain an analytic model for the true degree of conflict, as shown in equation (1). For such mixed parameters, we suggest Monte Carlo estimation.

(3) Pathology: Optionally, one may also test TM system performance under pathological transactions generated by EigenBench parameters. This can verify if the TM system is immune to the pathologies or susceptible to them.

(4) Explanation of TM application behavior: We believe that the above analysis should provide enough information to explain a certain TM application's performance behavior, as long as one knows its eigen-characteristics. However, one should also check if the application is governed by non-TM aspects (e.g. Amdhal limit, thread sync operation outside TX, etc.), which may not be explained by any TM characteristics.