TransPlant: A Parameterized Methodology For Generating Transactional Memory Workloads

James Poe[§], Clay Hughes[§], and Tao Li
Intelligent Design of Efficient Architecture Lab (IDEAL)
University of Florida, Gainesville, United States of America
{jpoe, cmhug}@ufl.edu, taoli@ece.ufl.edu

Abstract— **Transactional memory provides a means to bridge the discrepancy between programmer productivity and the difficulty in exploiting thread-level parallelism gains offered by emerging chip multiprocessors. Because the hardware has outpaced the software, there are very few modern multithreaded benchmarks available and even fewer for transactional memory researchers. This hurdle must be overcome for transactional memory research to mature and to gain widespread acceptance. Currently, for performance evaluations, most researchers rely on manually converted lock-based multithreaded workloads or the small group of programs written explicitly for transactional memory. Using converted benchmarks is problematic because they have been tuned so well that they may not be representative of how a programmer will actually use transactional memory. Hand coding stressor benchmarks is unattractive because it is tedious and time consuming. A new parameterized methodology that can automatically generate a program based on the desired high-level program characteristics benefits the transactional memory community.**

In this work, we propose techniques to generate parameterized transactional memory benchmarks based on a feature set, decoupled from the underlying transactional model. Using principle component analysis, clustering, and raw transactional performance metrics, we show that TransPlant can generate benchmarks with features that lie outside the boundary occupied by these traditional benchmarks. We also show how TransPlant can mimic the behavior of SPLASH-2 and STAMP transactional memory workloads. The program generation methods proposed here will help transactional memory architects select a robust set of programs for quick design evaluations.

Keywords: Transactional Memory, Workload Synthesis

I. INTRODUCTION

Transactional memory systems have received a lot of attention from both industry and the research community in recent years because it offers a way to ease the transition from programming for a single processing element to programming for many processing elements. The transactional memory (TM) model simplifies parallel programming by guaranteeing atomic execution for an entire block of code – a transaction. This eases the burden on the programmer who no longer needs to spend as much time reasoning about deadlocks and program invariants. However, parallel programming still bears the stigma of being tremendously difficult and burdensome to program correctly. So, even though programmers have several software-based transactional tools [1, 2] at their disposal, the production of valid transactional programs is almost non-existent. This forces researchers to convert lock-based or message-passing programs manually, which is itself exacerbated by the lack of a modern, cohesive, parallel benchmark suite. The dearth of representative, runnable, transactional memory programs increases the difficulty in developing and improving both hardware- and software-based transactional memory systems.

Fundamentally, designing a transactional memory system involves making decisions about its conflict detection, version management, and conflict resolution mechanisms, all of which can be implemented in software [3, 4, 5], hardware [6, 7], or a hybrid of the two [8, 9, 10]. Despite the increasing momentum in transactional memory research, it is unclear which designs will lead to optimal performance, ease of use, and decreased complexity. Further evaluation using a wide spectrum of transactional applications is crucial to quantify the trade-offs among different design criteria. To date, the majority of research into transactional memory systems has been performed using converted lock-based code or microbenchmarks. Because many of these benchmarks are from the scientific community, they have been optimized for SMP systems and clusters and represent only a fraction of potential transactional memory programs. Microbenchmarks are often too simplistic to stress increasingly large and complex multi-core designs and their interaction with the TM system. Several earlier studies [11-14] have shown that implementing a realistic application using transactional memory requires a clear understanding of the particular algorithm and the effort is non-trivial. Therefore, there is an urgent need for techniques and frameworks that can automatically produce representative transactional benchmarks with a variety of characteristics, allowing architects and designers to explore the emerging multi-core transactional memory design space efficiently.

The goal of this research is to develop mechanisms and methodologies that can automatically generate parameterized synthetic transactional workloads. Traditional synthetic benchmarks preserve the behavior of single- [15] or multithreaded [16, 17] workloads while the parameterized transaction synthesizer proposed in this paper is independent of any input behavior – capable of producing transactional code with widely varied behavior that can effectively stress transactional memory designs in multiple dimensions. This novel parameterized transaction framework can effectively 1) represent the heterogeneous concurrency patterns of a wide variety of applications and 2) mimic both the way that regular programmers use transactional memory and the way experienced parallel programmers can exploit concurrency opportunities. This allows architects and designers to explore large design spaces within which numerous design tradeoffs need to be evaluated quickly.

This paper makes the following contributions:

- TransPlant is the first automatic transactional benchmark synthesizer. TransPlant generates source code from a user-supplied feature set that, when compiled, can be run on real

This work is supported in part by NSF grant CNS-0834288, SRC grant 2008-HJ-1798, and by three IBM Faculty Awards. The authors acknowledge the UF HPC Center for providing computational resources.

§ Authors contributed equally to this work.

systems, simulators, or RTL modeling systems. The architecturally independent input parameters provide a robust method for generating programs for workspace exploration.

- We show how this methodology can be used to generate programs that represent all of the different high-level transactional characteristics giving researchers a much wider breadth of program selection.
- This paper shows how TransPlant can be used to replicate the behavior of existing transactional programs.

This paper is organized as follows. Section II provides a discussion of related research and Section III introduces TransPlant. Section IV describes our experimental methodology. Results are presented in Section V and the paper concludes in Section VI.

II. RELATED WORK

There are many benchmarks available for evaluating parallel computing systems, both traditional and transactional. Prior studies have attempted to quantify the redundancy in these and other frequently used application suites while other authors have proposed methods to reproduce the behavior of these programs using statistical models and workload synthesis. This section addresses how this previous research contributes to and reflects on our work.

A. Parallel Benchmarks

One roadblock that the TM/multi-core research and design community faces today is the lack of representative transactional memory benchmarks. As a result, a common practice in evaluating today's TM designs is to convert existing lock-based multithreaded benchmarks into transactional versions. There are several multithreaded benchmark suites to draw from: NPB [18], BioParallel [19], ALPBench [20], MineBench [21], SPEComp [22], SPLASH-2 [23], and PARSEC [24]. Most of these suites are domain specific (e.g. bioinformatics, multimedia, and data mining), which makes running all of the programs from one of these suites problematic. Of the above suites, only SPLASH-2 and PARSEC are not limited to a single application domain. Even so, converting many of these applications is not an attractive option because of complex libraries or threading models.

What is more, even a successful conversion does not mean that these programs are appropriate for use in a transactional memory evaluation. While these conventional multithreaded workloads may reflect the thread-level concurrency of transactional workloads to some extent, in many cases they have been heavily optimized to minimize the overhead associated with communication and synchronization. The fine-grain locking that these traditional programs exhibit does not represent the wide variety of expected behavior from transactional memory programs since any conversion leads to programs with infrequent transactions relative to the entire program. Much of the up-front benefit of transactional memory comes from its ease of use; programmers will be able to write parallel code bypassing much of the complex logic involved in providing correctness and minimizing time spent in synchronization regions. While programmers familiar with the pitfalls associated with parallel programming will be able to extract nearly the same performance out of transactions, those new to the field or those more deadline-over-performance oriented will be more interested in knowing that their code is correct and safe regardless of the size of the parallel regions and possible interactions.

B. Transactional Memory Benchmarks

Researchers have already begun thinking about how to test transactional memory systems and have developed microbenchmarks and applications to evaluate their behavior. The microbenchmarks used for these evaluations typically contain only a few small transactions making them too simple to stress increasingly large and complex multi-core designs. While these benchmarks are easily portable, they can be tedious to create and may not have any complex control flow, neither inter- nor intra-thread. To address the shortcomings of these microbenchmarks, a few real applications have been ported for use in transactional memory but these are stand-alone applications, many of which are not publicly available and their domain coverage is limited. Perfumo [12] and Minh [25] both offer transactional memory suites that attempt to expand this coverage. The problem with Perfumo's applications is that they are implemented in Haskell, making them extraordinarily difficult to port. On the other hand, Minh's contribution, STAMP, contains eight programs covering a wide range of applications and is written in C. But do these applications truly offer an expanded view of the transactional performance domain?

C. Benchmark Redundancy

Previous authors have shown that many programs within a benchmark suite exhibit tremendous amounts of redundancy [26, 27, 28, 29]. This is true of SPLASH-2, STAMP, and even the new PARSEC suite contains programs that not only share characteristics of the SPLASH-2 programs but also show some similarities with one another [24]. Computer architects need programs with widely varying behavior in order to evaluate design changes and some of these suites fall short. Shown below is an evaluation of STAMP and SPLASH-2 across a range of transactional features (the feature set is shown in Table 2). An overview of the mathematical processes involved in this evaluation can be found in Section IV-b.

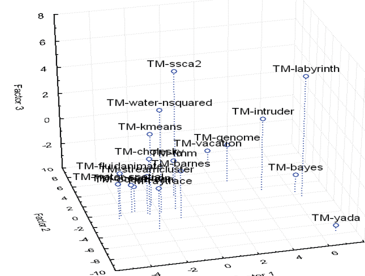

Figure 1. PC Plot of STAMP & SPLASH-2

Figure 1 is a plot of the first three principal components, which account for 64.6% of the total variance. Only 8 of the 18 benchmarks contain any real differences in their behavior in this domain. The rest of the benchmarks form a strong cluster, which indicates that many of the examined characteristics are similar if not the same. The hierarchal clustering (Figure 9) based on these three principal components shows the results more clearly. Beginning on the bottom with *labyrinth* and working up the dendrogram, one can see that the benchmarks beyond (and including in a relaxed interpretation) *fmm* and *genome* form relatively tight clusters. At a linkage distance of 4, 50% of the benchmarks have been clustered, showing that any evaluation of a transactional memory system using these benchmarks may not

Table 1. Transactional- and Microarchitecture-Independent Characteristics

Characteristic	Description	Values
Threads	Total number of threads in the program	Integer
Homogeneity	All threads have the same characteristics	Boolean
Tx Granularity	Number of instructions in a transaction	List, Normalized Histogram
Tx Stride	Number of instructions between transactions	List, Normalized Histogram
Read Set	Number of unique reads in a transaction	List, Normalized Histogram
Write Set	Number of unique writes in a transaction	List, Normalized Histogram
Shared Memory	Number of global memory accesses	List, Normalized Histogram (complete, high, low, minimal, none)
Conflict Distribution	Distribution of global memory accesses	List, Normalized Histogram (high, random)
Tx Instruction Mix	Instruction mix of transactional section(s)	Normalized Histogram (memory, integer, floating point)
Sq Instruction Mix	Instruction mix of sequential section(s)	Normalized Histogram (memory, integer, floating point)

stress all of the elements in its design and that new programs may be needed.

D. *Benchmark Synthesis*

Statistical simulation [16] and workload synthesis [15] capture the underlying statistical behavior of a program and use this information to generate a trace or a new representative program that maintains the behavior of the original program. This new representation has a reduced simulation time compared to the original application, making it ideal for coarse-grain tuning of early designs. Although most of this research has been on sequential programs, researchers have recently delved into multithreaded lock-based programs [16, 17]. Although this previous work does produce small fast-running programs, it differs from ours in that we do not use any program as a starting point. Our synthesis model works from an abstract input and the programs produced by TransPlant are built from the ground up using user-supplied inputs. This enables researchers to specify the program characteristics precisely in order to test abundant system aspects they want, similar to the work done by Joshi et al. [29] who showed how an abstract set of program characteristics could be used with machine learning to generate single-threaded stress benchmarks in the power domain.

III. TRANSPLANT

In the following section we describe the TransPlant model for generating transactional workloads and describe how it both differs from and expands upon currently available transactional benchmarks. We then detail its capabilities and end with a discussion on the implementation of the TransPlant framework.

A. *Design*

As long as there has been a need to quantify the behavior of a design using test workloads, there has been debate over the type of workload to use. Running real world applications has the advantage of providing designers with realistic inputs that may actually occur after production. However, running real applications also has substantial disadvantages. It can often be difficult to find real applications that cover a diverse design space, anticipate future workload patterns, and are easily executable on the system of choice. Moreover, while a diverse set of real applications can provide significant insight into overall, common case system performance, they can be inefficient at exploring the results of a specific design decision. Microbenchmarks, on the other hand, are much better suited to quickly assess the result of a specific execution pattern, however lack much of the context provided from real world applications. The goal of the TransPlant framework is to bridge the advantages of these two worlds within a transactional memory context. Using the TransPlant framework, a TM designer can efficiently construct a workload that is tuned precisely to the characteristics that he or she wishes to stress; starting either from a real application, or by using the tool to construct a design point that differs from any available workload.

B. *Capabilities*

The input to the TransPlant framework is a file describing the transactional characteristics the designer wishes to test and the output of the framework is a source file that can be compiled to produce a binary that meets those specifications. Table 1 describes the first order design parameters that the user can specify. Threads specify the total number of active threads while the Homogeneity flag indicates whether all threads will be homogeneous or whether the user will enumerate different characteristics for each thread. Transactional granularity specifies the size of the transaction with respect to instruction count and stride specifies the sequential distance between transactions. The Read Set and Write Set parameters describe the number of unique cache line accesses for reads and writes respectively, and the Shared Memory parameter describes the percentage of those locations that occur within shared memory regions. A key determinant of the overall transactional characteristics of a program is how the memory references are physically located within the transaction. The Conflict Distribution parameter indicates whether the shared memory references are evenly distributed throughout the transaction or whether a "high" conflict model is constructed where a read/write pair is located at the beginning and end of the transaction to maximize contention. Finally, the instruction mix of integer, floating point, and memory operations can be controlled independently for sequential and transactional portions.

A key feature of our input set is that while it covers most of the architecturally independent transactional characteristics, the level of granularity for which a user must specify the input set can be adjusted based upon what the designer is interested in. For example, most of the above inputs can be enumerated as a simple average, a histogram, a time-ordered list, or any combination thereof. Thus, if a designer is interested in an exact stride, alignment, or instruction count across threads and less interested in the read/write set sizes, the granularity and stride values can be defined in a time-sequenced list while the read/write set values are provided using a normalized histogram. This detailed level of control can prove invaluable in stressing a specific design implementation or in producing precise deterministic workloads to be used as a tool for debugging.

Finally, the framework allows for a "mimic mode" where a complete description of the program is provided as an input. When this mode is combined with a profiling mechanism, TransPlant can be used to reproduce a synthetic copy of an existing workload. This synthetic copy can be run in place of the original application (for example, in circumstances where the

original code is proprietary) or can be used as a baseline and modified to test how possible changes will affect future designs.

C. *Implementation*

The framework is comprised of four steps: input validation and construction of high-level program characteristics (skeleton), opcode and operand generation (spine), operand population (vertebrae), and code generation (program body). A high-level view of the framework is shown in Figure 2.

1) Validation and Skelton Creation: The first stage of benchmark generation within the TransPlant framework is to validate the input provided by the user. Since TransPlant accepts a wide variety of input formats (e.g. averages, lists, histograms, or any combination thereof), it is important that the input be validated to ensure that it describes a realizable binary. For example, since read set, write set, and transaction size can all be varied independently, TransPlant must validate each read set/write set combination to ensure there is a suitable transaction in which to fit the memory operations.

The first pass in the validation stage confirms that the user has specified all of the required options. Once all required options have been specified, the validation stage calculates the number of "Cells" required to represent the final binary described by the input. A Cell is the basic building block within the TransPlant framework and can be transactional, sequential, or synchronization. If any of the inputs provided by the user is in a list format, then the total number of cells is equal to the number of entries within that list. If the user provides all histogram inputs, TransPlant will calculate the minimum number of cells required to meet the histogram specifications perfectly (for example, if all normalized histogram inputs are multiples of 0.05 – then 20 cells can be used to meet the specifications).

Once the minimum number of cells has been instantiated, each cell is populated with values described by a list input or derived from a histogram input. In the case of histogram inputs, the cell lists are ordered based upon size and then the read set and write set values are populated from largest to smallest to ensure proper fitting. Other values, such as instruction mixes, shared memory percentages, and conflict distributions are randomly assigned based upon their histogram frequency.

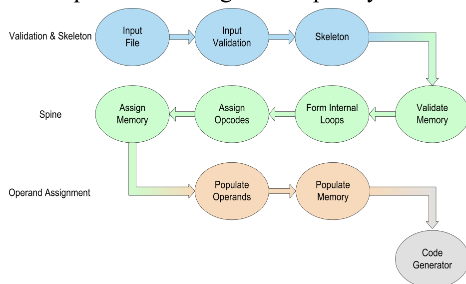

Figure 2. High-level Representation of TransPlant

2) Spine: Once the program contents have been validated, the cell list is sent to the next portion of the framework to generate a series of basic blocks derived from the individual cell characteristics. For each cell, the spinal column generator performs a second round of validation to ensure that it can meet the memory and size requirements of the cell. Because cells can be arbitrarily large, we attempt to form a loop within the cell. The loop must be able to preserve the instruction mix, shared memory distribution, and conflict distribution of the cell. The base value of the loop is determined by the number of unique memory references in the cell and is then adjusted to meet the remaining characteristics. A minimization algorithm is used to identify the optimal number of instructions to be included in the loop such that the remainder is as small as possible to control program size. This allows much more flexibility in terms of transaction stride and granularity without introducing much variation in the program. Once the cells have passed the second round of validation and any loop counters have been assigned, the spine generates opcodes for each instruction within the cell based on the instruction mix distribution. The last step in this phase attempts to privatize, localize, and add conflicts to the memory operations. The privatization mechanism assigns the memory type based on the number of shared reads and writes in each basic block by tagging the opcode as being private or global. Localization parses the memory references determining which ones should be unique (essentially building the read- and write-sets) and which ones reference a previous address within the same block. Memory conflictions are assigned based on the conflict distribution model and determines where each load and store within each block is placed.

3) Vertebrae: For each non-memory instruction, operands are assigned based on a uniform distribution of the registers, using registers t0-t5 and s2-s7 for non-floating-point operations and f2-f12 for floating-point operations. This ensures that the program contains instruction dependencies but does not tie the population to any specific input. For memory operations, we assign a stride value based on the instruction's privatization, localization, and confliction parameters. We maintain maps matching private and conflicted addresses for reuse to maintain the program's shared memory and conflict distribution models across threads. In addition, each instruction accesses memory as a stream – beginning with the base offset and walking through the array using the stride value assigned to it, restarting from the beginning when it reaches the boundary. The length of the array is predetermined based on the size of the private and global memory pools and the number of unique references in the program.

4) Code Generation: Because we rely on SESC [31] as our simulation environment, TransPlant was developed for the MIPS ISA but the backend can be decoupled for use with any ISA. The completed program is emitted in C as a series of header files, each containing a function for one of the program's threads. The main thread is written with a header containing initialization for the global memory as well as its own internal memory and variables. Both global and private memory are allocated using calls to *malloc()*. The base address of the memory pool is stored in a register, which along with offsets is used to model the memory streams. SESC uses the MIPS ISA and instructions within each thread are emitted in MIPS as assembly using the asm keyword, effectively combining the high-level C used for memory allocation with low-level assembly. To prevent the compiler from optimizing and reordering the code, the volatile keyword is used. The completed source code is then enclosed in a loop, which is used to control the dynamic instruction count for each thread.

This is primarily used to adjust the number of dynamic instructions required for the program to reach a steady state.

IV. METHODOLOGY

This section describes the variables used in our analysis. It also covers our data processing techniques: principal component and cluster analysis.

Table 2. Transaction Oriented Workload Characteristics

Program Characteristics		Synopsis
1	Transaction Percentage	Fraction of instructions executed by committed transactions.
2-11	Transaction Size	Total number of instructions executed by committed transactions stored in 10 buckets.
12-21	Read Conflict Density	The total number of potential conflict addresses read by a transaction divided by that transactions total read set stored in 10 buckets.
22-31	Write Conflict Density	The total number of potential conflict addresses written by a transaction divided by that transactions total write set stored in 10 buckets.
32-41	Read Set Size	Total number of unique memory addresses read by committed transactions stored in 10 buckets.
42-51	Write Set Size	Total number of unique memory addresses written by committed transactions stored in 10 buckets.

A. *Transactional Characterisitcs*

To characterize and compare transactional workloads, a set of features is needed that is largely independent of the underlying transactional model. It is important that these features are independent of the underlying transactional model because using metrics that are not (e.g abort rates and stall cycles) can result in widely varied outputs even when the same workload is run across different transactional dimensions (e.g. Eager Eager versus Lazy Lazy).

Table 2 describes the features we have found that play a dominant role in determining the runtime characteristics, contention, and interaction across transactional workloads. These features are used as inputs to the principle component analysis algorithm to classify the different transactional workloads. Our goal in choosing these metrics was to provide attributes that were able to describe the unique characteristics of individual transactions while remaining as independent of the underlying model as possible. Specifically, we record the *transaction percentage*, *transaction size, read-/write-set conflict densities*, and the *read-/write-set sizes* of each transaction. Since many transactional workloads exhibit heterogeneous transactions and different synchronization patterns throughout runtime execution, we wanted to provide a fine-grained analysis of the transactional characteristics throughout the program lifetime. To meet this goal, all but one of the characteristics is represented as a histogram, providing more information than a simple aggregate value.

The *transaction percentage* is the total number of retired committed transactional instructions divided by the total number of instructions retired. This ratio provides insight into how significant the transactional code was relative to the amount of total work completed. This metric is the only metric that is not a histogram. However, it is important as it helps to quantify the effect that the remaining characteristics have in the overall execution of a benchmark. For example, a workload that is comprised of transactions that are highly contentious but are only in execution for brief intervals may exhibit less actual contention than a workload comprised of fewer contentious transactions that occur with greater frequency. It is also important to note that we only consider committed and not aborted transactions within the transaction percentage. This is because while the amount of work completed or committed is largely determined by the workload and its inputs, aborted transactions are a function of the underlying architecture and can vary widely depending on architectural decisions.

Transaction size is defined as the total number of instructions committed by a transaction. This characteristic is comprised of a histogram describing the individual sizes of transactions across the entire execution time of a workload. This metric describes the granularity of the transactions across a workload. The granularity of a transaction is directly related to the period of time that a transaction maintains ownership over its read/write set, and thus helps to quantify the length of time that a transaction is susceptible to contention. It also provides insight into the amount of work that can potentially be lost on an abort, or the amount of time other transactions can be stalled on a NACK.

To assist in the characterization of contentious memory access patterns, we also include *read conflict density* and *write conflict density*. The *read conflict density* is defined as the total number of potentially contentious addresses within a transaction's read set divided by the total read set size of the transaction, and the *write conflict density* is defined as the total number of potentially contentious addresses within a transaction's write set divided by the total write set of the transaction. To calculate the addresses that can potentially result in contention within a transaction, we first run the entire workload and calculate the read/write sets for each transaction. Next, each memory address within a read set is marked as potentially contentious if any other transaction that was not located within the same thread wrote to that address. For addresses belonging to the write set, each memory address is marked as potentially contentious if any other transaction that was not located within the same thread either read or wrote to that address. Using this method, we can capture the worst-case contention rate of the read/write set for all possible thread alignments without the need to run exhaustive numbers of simulations. Note, however, that while this method is a conservative, worst case estimate of the contentiousness of a workload regardless of thread alignment, it is more accurate than simply identifying shared regions of memory as potentially contentious since it requires actual overlap of memory access patterns. Using this characteristic of a transaction, we are able to categorize the contentiousness of a specific transaction not simply based on the aggregate size of a memory set, but on the actual contentiousness of the memory locations within those sets.

While the *read/write conflict density* ratios are crucial in describing the underlying characteristics of individual read/write sets, they are unable to characterize the aggregate size of individual sets within a transaction. To meet this demand, we also include the *read set size* and *write set size* metrics, which quantify the number of unique memory addresses from which a program reads (*read set size*) as well as the number of unique memory addresses to which a program writes (*write set size*). The size of the read and write sets are important because they affect the total data footprint of each transaction as well as the period of time commits and aborts take.

When combined, the different transactional aspects that can be gathered from the characteristics described in Table 2 provide an

excellent means of quantifying the behavior of transactional workloads. However, due to the extensive nature of the data, a means of processing the data is necessary.

B. PCA and Hierarchical Clustering

Principal component analysis (PCA) is a multivariate analysis technique that exposes patterns in a high-dimensional data set. These patterns emerge because PCA reduces the dimensionality of data by linearly transforming a set of correlated variables into a smaller set of uncorrelated variables called principal components. These principal components account for most of the information (variance) in the original data set and provide a different presentation of the data, making the interpretation of large data sets easier.

Principal components are linear combinations of the original variables. For a dataset with p correlated variables ($X_1, X_2, \ldots, X_p$), a principal component Y_1 is represented as $Y_1 = a_{11}X_1 + a_{12}X_2 + \ldots + a_{1p}X_p$, where ($Y_1, Y_2, \ldots, Y_p$) are the new uncorrelated variables (principal components) and ($a_{11}, a_{12}, \ldots, a_{1p}$) are weights that maximize the variation of the linear combination. A property of the transformation is that principal components are ordered according to their variance. If k principal components are retained, where $k << p$, then $Y_1, Y_2 \ldots Y_k$ contain most of the information in the original variables. The number of selected principal components controls the amount of information retained. The amount of information retained is proportional to the ratio of the variances of the retained principal components to the variances of the original variables. By retaining the first k principal components and ignoring the rest, one can achieve a reduction in the dimensionality of the dataset. The Kaiser Criterion suggests choosing only the PCs greater than or equal to one. In general, principal components are retained so they account for greater than 85% of the variance.

Cluster analysis [30] is a statistical inference tool that allows researchers to group data based on some measure of perceived similarity. There are two branches of cluster analysis: hierarchical and partitional clustering. Our study uses hierarchical, which is a bottom-up approach that begins with a matrix containing the distances between the cases and progressively adds elements to the cluster hierarchy. In effect, building a tree based on the similarity distance of the cases. In hierarchical clustering, each variable begins in a cluster by itself. Then the closest pair of clusters is matched and merged and the linkage distance between the old cluster and the new cluster is measured. This step is repeated until all of the variables are grouped into a single cluster. The resulting figure is a dendrogram (tree) with one axis showing the linkage distance between the variables. The linkage distance can be calculated in several ways: single linkage (SLINK) defines the similarity between two clusters as the most similar pair of objects in each cluster and is the one used in this paper. Complete linkage (CLINK) defines similarity as the similarity of the least similar pair of objects in each cluster and average linkage (UPGMA) defines the similarity as the mean distance between the clusters.

V. Results

This section provides an evaluation of TransPlant using benchmarks generated to show program diversity as well as synthetic versions of the STAMP and SPLASH-2 benchmarks. For both sections, we measure the transactional characteristics of the new benchmarks and evaluate the results using principal component analysis and clustering. All benchmarks are run to completion with 8-threads using SuperTrans [33]. SuperTrans is built on SESC [31] and is a cycle accurate, multiple-issue, out of order common chip multiprocessor (CMP) simulator that supports

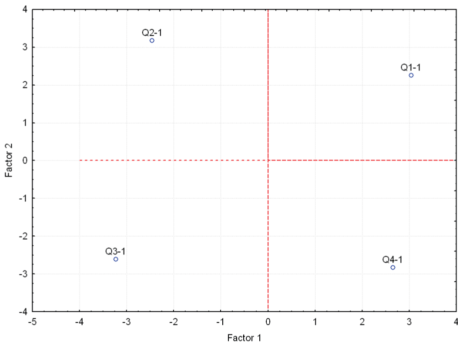

Figure 3. PC1-PC2 Plot of Synthetic Programs

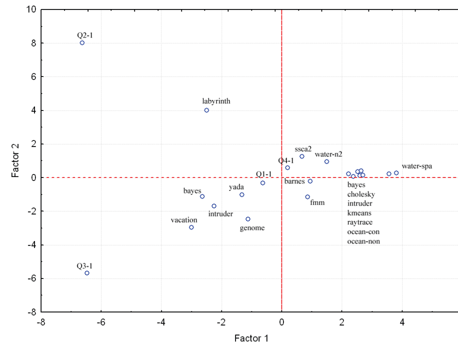

Figure 4. PC1-PC2 Plot of Unified PCA

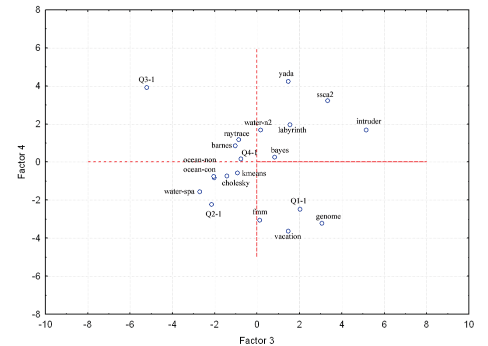

Figure 5. PC3-PC4 Plot of Unified PCA

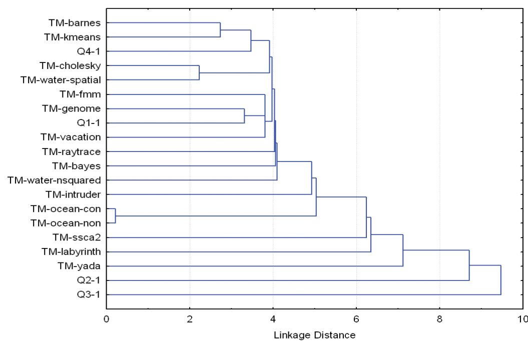

Figure 6. Dendrogram (Unified)

Table 3. Machine Configuration

Parameter	Value
Processor issue width	4
Reorder buffer size	104
Load/store queue size	72
Integer Registers	64
Floating Point Registers	56
Integer Issue Win Size	56
Floating Point Issue Win Size	16
L1 instruction cache size	32 KB
L1 data cache size	32 KB
L1 data cache latency	2
L2 cache size	4M
L2 cache latency	12

cycle accurate simulation of eager and lazy conflict detection and eager and lazy version management. Table 3 presents the microarchitecture configuration that was used for each core in the 8-core CMP simulation.

A. Stressing TM Hardware

In any evaluation, it is useful to be able to test a variety of design points quickly. To this end, TransPlant was used to generate a set of programs with widely varying transactional characteristics. These programs, *Q1-1* through *Q4-1*, represent the average behavior of each test quadrant. Figure 3 shows a plot of the first two principal components for the benchmarks generated here. These first two PCs account for 77.4% of the total variance. The first principal component is positively dominated by transactions sizes between 625 and 15k instructions; and negatively dominated by transactions larger than 390k instructions and read-/write-sets larger than 256 unique addresses. The second component is positively dominated by the extremes in write set (more than 1024 addresses) and read set (fewer than 2 unique addresses) and negatively dominated by the opposite extremes. Program *Q1-1* is comprised of transactions varying from 625 instructions to 78k instructions and read- and write-sets with 8 to 32 unique addresses. Program *Q2-1* is comprised of large transactions (between 390k and 976k instructions) with read- and write-sets ranging from 512 to 1024 unique addresses. Programs *Q3-1* and *Q4-1* are composed of large and small transactions, respectively, with read- and write-sets varying from 2 to 4 unique addresses for *Q4-1* and 64 to 128 addresses for *Q3-1*. Using the same variables, these programs were then compared to the benchmarks traditionally used to test transactional memory systems.

B. Workload Comparison

In this section, the overall program characteristics of the benchmarks generated in Section V-a, *Q1-1-Q4-1*, are compared with those of the SPLASH-2 and STAMP benchmarks. Specifically, we apply the same principal component analysis as above with the addition of the new benchmarks. With the reduced data from PCA, we use hierarchical clustering to group the benchmarks. We also evaluate the transactional performance of

Table 4. TM Workloads and their Transactional Characteristics (8Core CMP)

Benchmarks (input dataset)	Trans. Model	Trans. Started	Aborts	NACK Stalled Cycles (M)	Average Read Set Size*	Average Write Set Size*	Read/ Write Ratio	Avg. Commit Trans Length (Instructions)
barnes 16K particles	EE	70533	1554	2.33	6.71	6.53	1.07	204.09
	LL	69336	362	6.880302	6.71	6.53	1.07	204.10
fmm 16K particles	EE	45256	3	0.001771	13.43	7.34	1.82	175.60
	LL	45302	26	0.516338	13.43	7.34	1.82	175.52
cholesky tk15.O	EE	15904	19	0.015719	3.13	1.95	2.01	27.18
	LL	15963	78	0.057466	3.12	1.95	2.01	27.16
ocean-con 258x258	EE	2161	497	0.091549	3.00	0.27	12.93	10.39
	LL	1800	136	0.022013	3.00	0.26	13.44	10.38
ocean-non 66x66	EE	7200	5200	0.783498	3.00	0.38	9.79	13.25
	LL	2778	778	0.057183	3.00	0.36	10.22	13.17
raytrace Teapot	EE	141020	64279	22.43765	6.49	2.46	5.33	60.87
	LL	307376	230635	0.260170	7.49	2.46	6.51	73.54
water-nsq 512 molecules	EE	10398	22	0.002693	10.87	2.97	2.66	59.26
	LL	10482	106	0.654037	10.87	2.97	2.66	59.26
water-sp 512 molecules	EE	153	0	0.000146	2.48	1.37	1.68	133.25
	LL	226	73	0.003986	2.57	1.46	1.89	366.78
bayes 1024 records	EE	714	221	65.621712	151.65	77.62	1.95	80913.12
	LL	733	222	0.071399	154.78	80.63	1.91	84540.69
genome g256 s16 n16384	EE	6081	167	1.291080	35.78	9.62	3.71	2451.98
	LL	6195	281	1.156334	35.76	9.63	3.71	2452.32
intruder a10 l4 n2038 s1	EE	16658	5442	4.027422	14.02	8.84	1.58	494.65
	LL	18646	7430	0.434436	13.90	8.82	1.57	494.46
kmeans Random1000_12	EE	6710	5	0.014471	7.31	2.74	2.66	347.04
	LL	7075	370	0.044840	7.31	2.74	2.66	347.04
labyrinth 512 molecules	EE	382	174	323.617009	287.10	199.29	1.44	387340.10
	LL	694	486	0.048621	276.74	199.18	1.38	346683.35
ssca2 s11 i1.0 u1.0 l3 p3	EE	6758	32	0.013136	6.19	3.04	2.03	35.13
	LL	6941	45	0.075905	6.17	3.04	2.02	35.17
vacation 4096 tasks	EE	4096	0	0.036366	75.29	16.57	4.54	4558.53
	LL	4107	11	0.051667	75.29	16.57	4.54	4558.52
yada a20 633.2	EE	6573	1265	123.165861	55.85	26.84	2.08	16079.54
	LL	7247	1756	0.152548	54.16	25.35	1.93	14261.00
Q1-1	EE	1701	101	6.733200	22.0	20.69	1.05	7125.00
	LL	2660	1060	0.017968	22.0	20.69	1.05	7125.00
Q2-1	EE	1387	587	4271.581	627.2	622.36	1.38	1896093.75
	EL	3820	3020	0.116293	627.2	622.36	1.38	1896093.75
Q3-1	EE	1960	360	898.042	96.0	166.37	0.584	265625.00
	LL	4294	2694	0.0021559	96.0	166.37	0.584	265625.00
Q4-1	EE	1689	89	1.415997	3.20	3.60	1.085	958.41
	EL	2545	945	0.0032308	3.20	3.60	1.085	958.41

***Set size calculations based on 32B granularity**

the benchmarks across two different transaction designs.

1) Clustering: Figure 4 shows the first two principal components plotted against one another for all of the benchmarks. The first two principal components are largely dominated by the same characteristics described in Section V-a. However, there are more factors considered in this case and the first two components only comprise 47.1% of the total variance, changing factor weightings. Figure 4 shows programs *Q2-1* and *Q3-1* are separated from the rest of the benchmarks because they are comprised of medium to large transactions and have high contention. The PCA weights these variables more heavily in this evaluation. *Q1-1* and *Q4-1* are made up of transactions ranging from 5 to 625 instructions (with a very few large transactions) with moderate size read- and write-sets. Because their behavior is not skewed toward any particular feature in this domain, they fall in between the STAMP and SPLASH benchmarks.

Figure 5 shows principal components three and four plotted against one another. Factors three and four contain 24.6% of the variance with the third component positively dominated by small transactions and small write sets and negatively dominated by large write sets and small read sets. The fourth component is positively dominated by moderate read and write conflict ratios and large write sets and negatively dominated by moderate size transactions, read sets, and write sets. The program distribution here shows much stronger clustering because of the limited variance, but even so *Q3-1* and *Q2-1* stand out while *Q4-1* remains near the SPLASH programs and *Q1-1* maintains the same relative distance to *genome*, *fmm*, and *vacation*. The performance metrics in Section V-b2 confirm this behavior.

The clustering results in Figure 6 show *Q2-1* and *Q3-1* are the last in the amalgamation schedule and share the fewest program characteristics while *Q1-1* and *Q4-1* remain clustered with STAMP and SPLASH, showing that these programs share many of the inherent program characteristics of the traditional benchmarks. *Q1-1* through *Q4-1* show that TransPlant is capable is generating not only outlier programs but also programs with traditional performance characteristics. Further, if a cutoff value is used to choose a subset of programs able to represent the general behavior of all of the benchmarks [32], *Q2-1* and *Q3-1* are always included.

2) Performance: In order to validate the abstract characteristics discussed above, in this section we present the results of several transactional characteristics measured across the two primary hardware transaction models of Conflict Detection/Version Management, Eager/Eager and Lazy/Lazy respectively. The results are shown in Table 4. From this table it can be seen that while the synthetic benchmarks do not separate themselves in any single program characteristic, their metrics taken as a whole do differentiate them from the SPLASH and STAMP benchmarks. For example: while *Q2-1* is mostly comprised of very large transactions like *bayes* and *labyrinth* and has average read- and write-set sizes similar to *bayes*, it spends more time NACKing than any of the other programs and is about average in the number of aborts that it experiences. What is more, when the differences between EE and LL are examined, it can be seen that *Q2-1* behaves more like labyrinth and *Q3-1* behaves similarly but with much smaller read- and write-sets. In the above clustering, *Q1-1* was clustered with genome (loosely). In this case, they are both comprised of transactions that vary greatly in size, skewing the average length. Because they share this layout, their read and write conflict ratios are very similar. This also explains *Q4-1*, whose read/write ratio resembles that of barnes but whose general read set behavior is more closely related to cholesky. This shows that the tool is able to produce programs with vastly different high-level characteristics but can maintain a realistic representation of program behavior.

Table 5. Abort-Transaction Ratios

Benchmarks	AbortCycles/ TotalCycles	Avg. Commit (Instructions)	AbortCycleRatio/ TransactionSize
barnes	4.88E-03	2.04E+02	2.39E-05
bayes	1.22E-02	4.49E+05	2.73E-08
cholesky	1.90E-06	2.72E+01	6.99E-08
fmm	2.00E-07	1.76E+02	1.14E-09
genome	2.11E-02	1.20E+03	1.76E-05
intruder	4.65E-01	4.96E+02	9.38E-04
kmeans	3.40E-06	1.00E+02	3.39E-08
labyrinth	9.19E-01	5.18E+05	1.78E-06
ocean-con	2.45E-05	1.04E+01	2.36E-06
ocean-non	1.96E-03	1.33E+01	1.48E-04
raytrace	9.25E-02	6.09E+01	1.52E-03
ssca2	4.58E-04	3.40E+01	1.35E-05
yada	2.27E-01	1.46E+04	1.56E-05
testCase	8.37E-01	1.00E+01	8.37E-02

C. Case Study: Abort Ratio and Transaction Size

To show how TransPlant can be used to generate evaluation programs that are of interest to a designer but are unavailable in current benchmarks, *testCase*, was created. Using TransPlant, the development time for the benchmark was less than 10 minutes. The goal in creating this benchmark was to highlight contention, which from a design point of view is one of the most interesting characteristics of a transactional program. And, while it is relatively easy to force contention in very large transactions, without synchronization mechanisms it is difficult to create contention with small transactions. Although most benchmark studies report contention, it is almost never evaluated with respect to the granularity of the transactions. This is particularly important because previous research [33] has shown that highly contentious fine grain transactions offer the most room for optimization and are representative of the types of non-scientific database-driven applications or compiler optimized applications that TM will be applied to in the future.

To associate transaction size with abort time, we use aborted cycles to total cycles to average transaction size. Table 5 shows the results when we compare *testCase* to the STAMP and SPLASH benchmarks. *testCase* is a fully synthesized workload created using the TransPlant framework with high contention and transaction sizes limited to 10 instructions. Even with the workload limited to very fine-grain transactions, this program spends nearly as much of its execution time aborting as *labyrinth*, whose average transaction size is over 500k instructions; moreover its abort-transaction size ratio is nearly two orders of magnitude larger than the next contender, *raytrace*.

D. Benchmark Mimicry

While being able to create benchmarks based on an arbitrary input is useful for testing and debugging, it is important that the

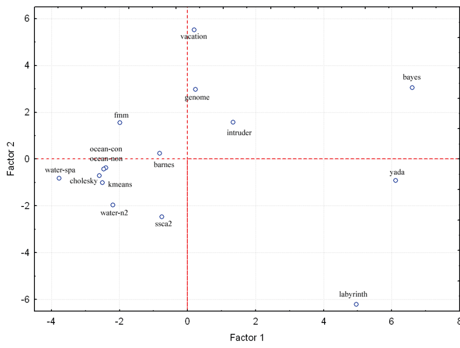

Figure 7. PC1-PC2 Plot of Original Applications

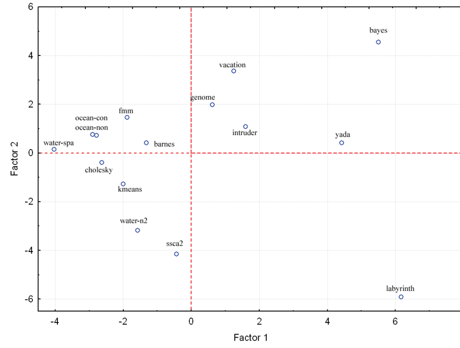

Figure 8. PC1-PC2 Plot of Synthetic Applications

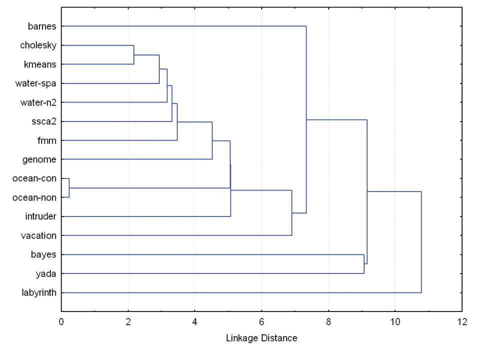

Figure 9. Dendrogram From Original Cluster Analysis

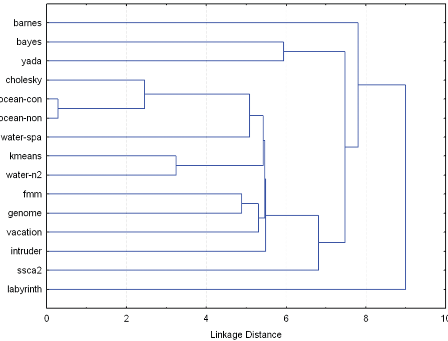

Figure 10. Dendrogram From Synthetic Cluster Analysis

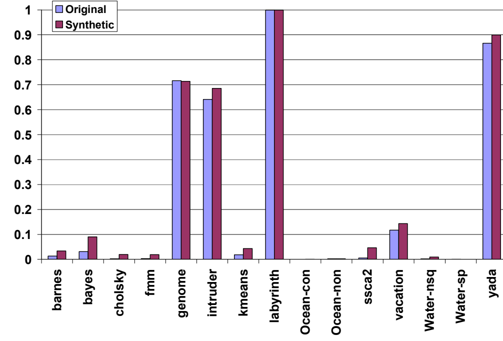

Figure 11. $\dfrac{AbortCycles + NACKCycles + CommitCycles}{TotalCycles}$

tool be able to replicate the behavior of existing benchmarks. In this section, we use PCA and clustering to show how our tool can use a trace to generate a synthetic benchmark that maintains the high-level program characteristics of the SPLASH and STAMP benchmarks.

Figure 7 shows the plot of the first two principle components of the STAMP and SPLASH benchmarks using the inputs from Table 2. These two factors comprise 48.9% of the total variance. Figure 8 shows the same plot of the first two factors of the synthetic representation, representing 33.4% of the variance. While these figures match almost perfectly, there is some deviation brought about by the read- and write-conflict ratios. These are calculated using an absolute worst-case estimate, as described in Section IV-a. When the profiler generates the input for the tool, it has best case of the actual contentious memory addresses, producing a less conservative, more accurate, representation. Figure 9 shows the hierarchical clustering for the original applications and Figure 10 shows the clustering for the synthetic representation. While the amalgamation schedule is slightly off, the overall representation is almost exact.

Finally, Figure 11 shows the ratio between total transactional cycles (aborts+NACKs+commits) and the total completed cycles of the original and synthetic benchmarks when run on SuperTrans. This metric is of particular significance because transactional cycles include both those cycles due to committed work (i.e. real work completed) as well as cycles wasted in contentious behavior (e.g. aborted transactions, NACK stall cycles, commit arbitration, etc). While much of the committed work is within our direct control in the synthetic benchmark creation, the contentious behavior is a ***result*** of the workload's interaction with the transactional model. From Table 4, it can be seen that for many of the benchmarks these contentious cycles account for a significant portion of the transactional work. Thus, while the PCA results provide validation that our synthetic benchmarks are able to preserve the architecture independent workload characteristics of the original benchmarks, Figure 11 clearly shows that the synthetic benchmarks also preserve the alignment and fine-grain behavior of the original benchmarks.

VI. Conclusion

The progression from single processing elements to multiple processing elements has created a gap in the performance gains offered by new generations of chips. Without the software available to exploit potential task- and data-parallel performance gains, many of the chip's resources remain idle. This software deficiency is partially due to the difficulty in developing parallel applications. Transactional memory may be able to help ease some the difficultly by providing programmers an easy-to-use interface that guarantees atomicity. But, transactional memory researchers are faced with the task of developing hardware and software solutions for an *emerging* programming paradigm, necessitating the use of conventional multithreaded programs as a starting point.

Converting the SPLASH-2 suite to use transactions is an easy way to bridge the gap between traditional locks and transactions, but this is because these programs have been so heavily optimized; such a limited feature set is eclipsed by the possibilities that transactional memory offers. The STAMP suite, while written explicitly for transactional memory, provides a more robust set of programs but ties the user to a limited set of inputs. We wanted to bridge this feature gap and provide researchers with a means to quickly generate programs with the

features important to their research without relying on external programs of which only a portion of the entire execution may be interesting.

Using principle component analysis, clustering, and raw transactional performance metrics, we have shown that TransPlant is capable of creating programs with a wide range of transactional features. These features are independent of the underlying transactional model and can be tuned in multiple dimensions, giving researchers the freedom they need in testing new transactional memory designs. In addition, we have shown how TransPlant can use profiling information to create synthetic benchmarks that mimic the high-level characteristics of existing benchmarks. This allows for the creation of equivalent transactional memory programs without manually converting an existing program and provides a venue for the dissemination of possibly proprietary benchmarks without dispersing the source code. The framework presented in this paper provides a limitless number of potential transactional memory programs usable by transactional memory architects for quick design evaluations.

REFERENCES

[1] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. S. III, and M. L. Scott, Lowering the Overhead of Non-blocking Software Transactional Memory, In ACM SIGPLAN Workshop on Transactional Computing, 2006.

[2] Intel® C++ STM Compiler, Prototype Edition 2.0. http://software.intel.com/

[3] D. Dice, O. Shalev & N. Shavit, Transactional Locking II, In Proc. of the International Symposium on Distributed Computing, 194-208, 2006.

[4] T. Harris and K. Fraser, Language Support for Lightweight Transactions, In Proc. of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2003.

[5] B. Saha, A.-R. Adl-Tabatabai, et al., A High Performance Software Transactional Memory System for A Multi-Core Runtime, In Proc. of the International Symposium on Principles and Practice of Parallel Programming, 2006.

[6] L. Hammond, V. Wong, et al., Transactional Memory Coherence and Consistency, In Proc. of the International Symposium on Computer Architecture, 2004.

[7] K. E. Moore, J. Bobba, et al., LogTM: Log-Based Transactional Memory, In Proc. of the International Conference on High-Performance Computer Architecture, 2006.

[8] C. Cao Minh, M. Trautmann, et al., An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees, In Proc. of the International Symposium on Computer Architecture, 2007.

[9] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson, Architectural Support for Software Transactional Memory, In Proc. of the International Symposium on Microarchitecture, 2006.

[10] A. Shriraman, M. Spear, et al., Integrated Hardware-Software Approach to Flexible Transactional Memory, In Proc. of the International Symposium on Computer Architecture, 2007.

[11] I. Watson, C. Kirkham and M. Lujan, A Study of a Transactional Parallel Routing Algorithm, In Proc. of International Conference on Parallel Architecture and Compilation Techniques, 2007.

[12] C. Perfumo, N. Sonmez, A. Cristal, O. S. Unsal, M. Valero, T. Harris, Dissecting Transactional Executions in Haskell, In ACM SIGPLAN Workshop on Transactional Computing, 2007.

[13] J. Chung, et al., The Common Case Transactional Behavior of Multithreaded Programs, In Proc. of International Conference on High Performance Computer Architecture, 2006.

[14] M. Scott, et al., Delaunay Triangulation with Transactions and Barriers, In Proc. of the International Symposium on Workload Characterization, 2007.

[15] L. Eeckhout, R. Bell Jr., B. Stougie, K. De Bosschere, and L. John, Control Flow Modeling in Statistical Simulation For Accurate And Efficient Processor Design Studies, In Proc. of International Symposium on Computer Architecture, 2004.

[16] S. Nussbaum, S. and J. E. Smith, Statistical Simulation of Symmetric Multiprocessor Systems, In Proc. of Annual Simulation Symposium, 2002.

[17] C. Hughes and T. Li, Accelerating Multi-core Processor Performance Evaluation using Automatic Multithreaded Workload Synthesis, In Proc. of International Symposium on Workload Characterization, 2008.

[18] H. Jin, M. Frumkin, et al., The OpenMP Implementation of NAS Parallel Benchmarks And Its Performance. Technical Report, 1999.

[19] A. Jaleel, M. Mattina, et al., Last Level Cache Performance of Data Mining Workloads on a CMP – A Case Study of Parallel Bioinformatics Workloads, In Proc. of International Symposium on High-Performance Computer Architecture, 2006.

[20] M. Li, et al., The ALP Benchmark Suite For Complex Multimedia Applications, In the Proc. of International Symposium on Workload Characterization, 2005.

[21] R. Narayanan, et al. Minebench: A Benchmark Suite For Data Mining Workloads. In Proc. of International Symposium on Workload Characterization, 2006.

[22] Standard Performance Evaluation Corporation, SPEC OpenMP Benchmark Suite. http://www.spec.org/omp.

[23] S. C. Woo, et al., The SPLASH-2 Programs: Characterization and Methodological Considerations, In Proc. of International Symposium on Computer Architecture, 1995.

[24] C. Bienia, S. Kumar, and K. Li, PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors, In Proc. of International Symposium on Workload Characterization, 2008.

[25] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, STAMP: Stanford Transactional Memory Applications for Multi-Processsing, In Proc. of International Symposium on Workload Characterization, 2008.

[26] R. H. Saavedra and A. J. Smith, Analysis of Benchmark Characteristics and Benchmark Performance Prediction, ACM Trans. Computer Systems, 1996.

[27] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, Quantifying The Impact of Input Data Sets On Program Behavior and Its Applications, Journal of Instruction Level Parallelism, 2003.

[28] L. Eeckhout and K. De Bosschere, Hybrid Analytical-Statistical Modeling for Efficiently Exploring Architecture and Workload Design Spaces, In Proc. of Parallel Architectures and Compilation Techniques, 2001.

[29] A. Joshi, L. Eeckhout, L. John, and C. Isen. Automated Microprocessor Stressmark Generation, In Proc. of High Performance Computer Architecture. 2008.

[30] C. H. Romesburg, Cluster Analysis for Researchers, Lifetime Learning Publications, 1984.

[31] SESC: A Simulator of Superscalar Multiprocessors and Memory Systems With Thread-Level Speculation Support, http://sourceforge.net/projects/sesc

[32] J. Yi, R. Sendag, L. Eeckhout, A. Joshi, D. Lilja, and L. John, Evaluating Benchmark Subsetting Approaches, In Proc. of International Symposium on Workload Characterization, 2006.

[33] J. Poe, C. Cho, and T. Li, Using Analytical Models to Efficiently Explore Hardware Transactional Memory and Multi-core Co-design, In Proc. of International Symposium on Computer Architecture and High Performance Computing, 2008.