# MULTICORE PROCESSOR AND HARDWARE TRANSACTIONAL MEMORY DESIGN SPACE EVALUATION AND OPTIMIZATION USING MULTITHREADED WORKLOAD SYNTHESIS

By

CLAYTON M. HUGHES

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2010

To Bubba and "D"

ACKNOWLEDGMENTS

The decade since I returned to school has been both exhilarating and harrowing. I could not have made it with the continual support, understanding, and expertise of my friends, family, and colleagues.

It has been a great honor to have Dr. Tao Li as my advisor. He said Yes when he didn't have to and gave me the freedom to pursue my research. He gave me confidence and pushed me when no one else would. His unflagging support has made this work possible.

I would like to thank the institutions that made this research possible: the University of Florida, IBM, and the Global Research Corporation. I would like to thank my advisory committee, Dr. Shigang Chen, Dr. Ann Gordon-Ross, and Dr. Jih-Kwon Peir, for taking the time to give me feedback and improve this work.

I am deeply indebted to the other members of the Intelligent Design of Efficient Architectures Laboratory (IDEAL) especially Wangyuan Zhang and my coauthor James Poe, whose argumentativeness helped me solidify my ideas and expand my research. I want to thank Daniel Durnbaugh and Gerard Virga for always being home and mnx in #tbar for knowing how to spell.

I would like to thank my parents, Noah and Jan Legear, for understanding when four day visits turned into two and my sister, Haley, for giving me a bed and niece. But I owe my biggest debt of gratitude to my grandparents, Max and Dorothy Bowden, to whom this work is dedicated. I cannot even begin to summarize the impact that they had on my life. Finally, to "Mama" Jimmie Prestwood whose thought always reminds me of the quote, "Curiosity, like coffee, is an acquired need. Just a titillation at the beginning, it becomes with training a raging passion."

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

# MULTICORE PROCESSOR AND HARDWARE TRANSACTIONAL MEMORY DESIGN SPACE EVALUATION AND OPTIMIZATION USING MULTITHREADED WORKLOAD SYNTHESIS

By

Clayton M. Hughes

December 2010

Chair: Tao Li
Major: Electrical and Computer Engineering

The design and evaluation of microprocessor architectures is a difficult and time-consuming task. Although small, hand-coded microbenchmarks can be used to accelerate performance evaluation, these programs lack the complexity to stress increasingly complex architecture designs. Larger and more complex real-world workloads should be employed to measure the performance of a given design and to evaluate the efficiency of various design alternatives. These applications can take days or weeks if run to completion on a detailed architecture simulator. In the past, researchers have applied machine learning and statistical sampling methods to reduce the average number of instructions required for detailed simulation. Others have proposed statistical simulation and workload synthesis, which can produce programs that emulate the execution characteristics of the application from which they are derived but have a much shorter execution period than the original. However, these existing methods are difficult to apply to multithreaded programs and can result in simplifications that miss the complex interactions between multiple concurrently running threads.

This study focuses on developing new techniques for accurate and effective multi-threaded workload synthesis for both lock-based and transactional memory programs. These new benchmarks can significantly accelerate architecture design evaluations of multicore processors. For benchmarks derived from real applications, synchronized statistical flow graphs that incorporate inter-thread synchronization and sharing behavior to capture the complex characteristics and interactions of multiple threads are proposed along with a thread-aware data reference model and a wavelet-based branch model to generate accurate memory access and dynamic branch statistics. Experimental results show that a framework integrated with the aforementioned models can automatically generate synthetic programs that maintain characteristics of original workloads but have significantly reduced runtime.

This work also provides techniques for generating parameterized transactional memory benchmarks based on a statistical representation, decoupled from the underlying transactional model. Using principle component analysis, clustering, and raw transactional performance metrics, it can be shown that TransPlant can generate benchmarks with features that lie outside the boundary occupied by these traditional benchmarks. It is also shown 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 in both the power and performance domains.

# CHAPTER 1
# INTRODUCTION

The entire microprocessor industry is moving towards multi-core architecture design. To take full advantage of multi-core CPU chips, computer workloads must rely on thread-level parallelism. Software engineers use multiple threads of control for many reasons: to build responsive servers that communicate with multiple parallel clients, to exploit parallelism in shared-memory multiprocessors, to produce sophisticated user interfaces, and to enable a variety of other program structuring approaches. Multi-threaded programming has been widely exploited in the construction of real-world applications spanning everything from scientific simulation to commercial applications. With the ongoing language and library (e.g. Java, C#, OpenMP, C++/C-Pthreads and Win32 threading APIs) design efforts, multi-thread running on multi-core hardware is likely to be the prevalent execution paradigm for the next generation of computer systems.

The design, evaluation, and optimization of multi-core architectures present a daunting set of challenges. The complexity of today's uni-core processors results in many hundreds or thousands of tradeoffs being evaluated in the early, high-level design phases. It is well known within the processor architecture design community that examining complex real-world applications using detailed performance models is impractical. The design space exploration of multi-core architectures is likely to be even more prohibitively expensive. Not only the configuration of individual cores, but also the interaction between cores (e.g. shared/private caches, coherency protocols, interconnection topology, and quantity/heterogeneity of multiple cores) needs to be examined. To compound this problem, as the number of cores and the complexity of

their interconnects increase, simulations become even slower. For example, compared with a simulator that models a uni-core processor, a 16-core chip multiprocessor simulator can slow down the simulation speed by as much as 60x [58]. This trend will be even more pronounced for simulating future multi-core architectures, which are predicted to have an even a larger number of cores. Due to the large simulation overhead of multi-core architectures, those explorations and optimizations cannot be pursued without developing techniques and tools that allow designers and researchers to rapidly examine numerous design alternatives for this emerging architecture paradigm. But as processors move further into the multicore era, a shift in programming focus will be required to extract the benefits of these new resources; transactional memory may be part of this paradigm shift.

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 [47] [31] 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 [15] [25] [65], hardware [24] [39], or a hybrid of the two [51] [66] [74]. 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 [82] [59] [13][70] 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.

# CHAPTER 2
# ACCELERATING MULTI-CORE PROCESSOR DESIGN SPACE EVALUATION USING AUTOMATIC MULTI-THREADED WORKLOAD SYNTHESIS

## Background and Motivation

To accelerate multi-core design evaluation, innovative techniques and methodologies are proposed for creating synthetic multi-threaded workloads with significantly reduced runtime. Applying techniques from statistical simulation to these elements enables the generation of accurate workload characterizations and produces a synthetic workload comprised of the dynamic execution features of the original multi-threaded program. Statistical flow graphs, proposed by Eeckhout et al. [16], are extended to include thread interactions. Moreover, novel thread-aware data reference models and wavelet-based branching models are developed to capture complex multi-threading memory access behavior and architectural independent dynamic branch characteristics. A walk of synchronized statistical flow graphs augmented with the proposed novel memory and branching models automatically produces a synthetic program emitted as a series of low-level statements embedded in a C program. When compiled, the synthetic program maintains the dynamic runtime characteristics of the original program but with far fewer instructions and significantly reduced runtime. Because the miniature program can be compiled into a binary, it can execute on a variety of platforms making it ideal for many aspects of architecture design.

## Workload Synthesis for Efficient Microprocessor Design Evaluation

The prohibitively long simulation time in processor architecture design has spurred a burst of research in recent years to reduce this cost. Among those, workload synthesis [3][27][36] has been shown to be an effective methodology to accelerate architecture design evaluation. The goal of this approach is to create reduced miniature

benchmarks that represent the execution characteristics of the input applications but have a much shorter execution period than the original applications.

From the perspective of architectural design evaluation, it is essential that the synthetic program efficiently and accurately model the behavior of the original application. Prior studies [3][27][36] focus exclusively on sequential benchmark synthesis. While multiple independent sequential programs can be used to study system throughput, and parallel execution of sequential programs provides some information, multi-threaded applications perform quite differently from sequential programs executed in a multi-programmed manner. Threads coordinate and synchronize with one another to produce correct computation results. The interactions between threads impose a global order on instructions and events. Threads read and write shared variables in the memory hierarchy, generating additional cache misses and coherency traffic. These features result in design decisions that are significantly different from those made based on multiple sequential program execution

As an example, consider the program shown in Figure 2-1. This very simple program generates two children, each of which attempt to execute the function myFunction(), and then waits for both threads to finish their work. All of the operations in myFunction() are enclosed in a lock/unlock pair to ensure that only a single thread is allowed access to the operations that modify the global shared variable, myUnsigned. Even this small program is capable of exposing the difficulties involved in attempting to use multiple single-threaded programs to mimic the behavior of a multi-threaded program. The thread management functions, pthread_create() and pthread_join(), and synchronization functions, mutex_lock() and mutex_unlock(), imply timing within the

code. A concatenation of the three threads, forming a single-threaded program, or even generating three separate programs obfuscates or loses this timing information. In this work, a methodology to preserve this information is proposed and encoded into a synthetic representation of the original program.

## Proposed Multi-threaded Workload Synthesis Techniques

Our proposed multi-threaded workload synthesis techniques consist of three primary steps: workload characterization, building and pruning statistical flow graphs, and synthetic code generation. Because workload characterization and statistical flow graph generation are so tightly coupled, they are included together in the discussion below.

### Multi-threaded Workload Representation

The statistical flow graph (SFG) proposed in [16][2] are extended to characterize a multi-threaded program's dynamic execution at the basic block level. In a SFG each node represents a unique basic block and is annotated with the corresponding execution frequency. An edge in the SFG represents a branch annotated with taken/not-taken probability. A basic block-level profiling of the original program is performed to record a sequence of instructions within each basic block. If there is interaction with a threading library, the basic block is augmented with additional information (such as the starting address of a spawn thread in the case of thread creation). The above information is integrated into synchronized statistical flow graphs (SSFG), which capture the statistical profile of both individual and interacted threads.

Figure 2-2 illustrates an example of using the proposed synchronized statistical graphs to represent a program containing three separate threads. In Figure 2-2, T0 is the main thread and T1 and T2 are two child threads. The graphs that are generated for

each thread are annotated to include transition probabilities between each node in the graph as well as inter-thread synchronization and sharing patterns. As can be seen, a separate statistical flow graph is generated for each of the threads. The edges are weighted according to the transition probabilities derived from the original program. The hashed nodes in T0, B and C, represent thread control points. In this case, T1 is spawned in node B and T2 is spawned in node C. Additionally, any potentially shared data is encoded with the nodes. T1 and T2 have two separate critical sections that were indicated as explicitly shared in the original program, node F from T1 and node N from T2 (protected by lock L1) and nodes G and I from T1 and node M from T2 (protected by lock L2). These SSFGs provide a profile of the dynamic execution of each thread, exposing the effects of synchronization and control flow between the threads.

**Statistical Flow Graph Reduction**

Once a synchronized statistical flow graph is created for each thread, the graph reduction factor method proposed by Eeckhout et al. [16] is applied to reduce node instances in the statistical flow graph. For each node in the graph, its instance count is divided by *R* where *R* is defined as the graph reduction factor, so that the new instance counts are a factor *R* smaller than the original. If the new instance count is less than one, the node and all in- and out-edges are pruned from the graph. This ensures that only frequently executed basic blocks within the original workload are considered when generating the synthetic code.

Because nodes are removed from the original SFG, the reduced representation can become disconnected. While previous research ignored the disconnected portions of the graph, in this study all nodes remaining after the reduction factor has been applied are retained and available for inclusion in the synthetic. Currently, the

appropriate *R* is derived experimentally. Finding a heuristic that can be used to determine the optimal reduction factor is left for future work.

### Code Generation

Once the reduced statistical flow graphs are created, the methods proposed for sequential workload synthesis [3][36] are used to instantiate low-level instructions enveloped in a traditional C program. The synchronization primitives and thread-related events such as create, join, detach, etc. are emitted as assembly language macros and low-level system calls, utilizing the interface provided by glibc and the OS. More details on synthetic benchmark generation can be found in Section 2-2.5.

## Automatically Synthesizing Multi-threaded Workloads

SSFG construction and reduction methods are implemented as described in Section 2-3. The framework consists of three components: front-end instrumentation, program flow analysis, and code generation. Details about each phase are discussed below.

### The Front End

The front-end of the automatic multi-threaded workload synthesis framework is implemented using the Intel Pin tools [45], a dynamic instrumentation system capable of capturing the execution of an application by inserting customized code at key program locations. A disassembler is used to identify call sites for multi-threading primitives in the pthread library and pass these addresses to the Pin tool. The tool monitors the number of times a basic block is executed and its component instructions, whether a branch is taken or not, and each instruction’s data reference locality. If any calls are made to a threading library, these events are categorized and associated with the calling block.

For each basic block, a list of its instructions is recorded and its starting address is used as a node identifier to build a dynamic CFG. Each basic block is inserted only once; if it is encountered again, its occurrence count is incremented. Edges are inserted into the graph in a similar fashion; new edges are added when nodes are added, otherwise their occurrence count is incremented. The tail of each basic block is checked to see whether the branch was taken or not taken and the result is stored as a unique bit vector for each basic block. The front end also collects information for routines within a target binary, specifically the threading library functions used for control, such as pthread_create() and pthread_destroy(). When one of these control points is identified, the corresponding node is tagged according to the type of control action. Profiling is also carried out at the instruction level so that paired function calls, such as lock/unlock, can be identified by their calling address. Identifying when the program enters and exits these functions allows the framework to capture portions of the user code intended to be synchronized with other threads.

Overhead incurred during runtime has been minimized to reduce the effects that profiling has on the timing of multi-threaded programs [1]. To help achieve this minimization, extensive use of the efficient data structures provided by the Boost library [8] is made to manage the graphs. While the framework is implemented as a customized Pintool, only the front end utilizes the Pin Instrumentation Library and very little analysis is performed at runtime. This makes the framework portable to other instrumentation tools or simulation environments.

**Thread-aware Memory Reference Model**

A thread-aware memory reference model is proposed to capture original program's data reference locality. While prior work [23][55] based their memory models

on a program's cache and TLB miss rates, the framework models the stride of the effective addresses touched by the original program. Thus, it captures programs' inherent memory access locality independent of microarchitecture implementations. The model distinguishes itself from previous stride-based memory models [3][36] in that it consists of two independent parts: thread-private and thread-shared.

Private memory accesses are assumed to be any reference that occurs outside of a critical section (not including read-only shared data accesses) and any reference within a critical section that is only touched by the current thread. The private memory portion of the memory model maintains separate stride information for memory reads and memory writes. For each memory read, the stride between successive references is recorded and the result is stored in a histogram. Memory writes are handled the same way and stored in a separate histogram. These histograms maintain counts for six stride values: 1-, 2-, 4-, 8-, 16-, 32-, and greater than 32-bytes. At analysis time, a cumulative distribution of the stride values is generated for each thread and used during the generation of the synthetic program to generate a circular stream of memory references.

Shared memory accesses are recorded when any read or write within a critical section touches a portion of memory touched by another thread. Data for shared memory references is stored at the instruction level as opposed to the thread level. When an instruction accesses a shared memory location for the first time, the effective address is recorded and a list is started that records the effective address for all successive memory references by that instruction. At analysis time, this information is converted to a cumulative distribution for the stride pattern of the instruction. This

distribution is stored with the instruction and the first reference address for use during code generation. If this instruction is encountered during code generation, a search is performed for any shared-memory instruction with an effective address within 32 bytes. These instructions are then matched to a common starting point within the allocated shared memory and successive references to these locations are based on the stride pattern.

Figure 2-3 provides an example of how the memory model translates high-level memory references to low-level assembly. In the sample code fragment, there are three variables: u_int_1 and array_1, which are private, and myUnsigned, which is shared. During profiling, the starting address is recorded for the three shared memory references along with the stride of the next reference for each instruction. For the private references, separate write- and read-stride distributions are maintained for each thread. At code generation time, the starting addresses for the three shared references are matched to one another and the base is inserted. If there are subsequent traversals of this basic block, the memory reference will change based on the stride distribution. In the example, the address will never change since there was never an offset in the effective address. The thread-private data references are assigned strides based on the cumulative read and write stride distributions for the thread. Memory operations are then inserted into the synthetic with the stride offset. In the example, all of the memory operation access integer values at four-byte intervals.

**Flow Analysis**

As mentioned in Section 2-4.1, to reduce perturbations in the system, which can influence the behavior of a multi-threaded program [1], only minimal analysis is performed at run time. The majority of the analysis is performed offline by parsing the

results and augmenting the control flow graph with additional information. The final output of this offline analysis is a series of statistical flow graphs like the ones shown in Figure 2-2. Offline analysis consists of five steps: computing edge weights, identifying child processes (threads), graph reduction, branch modeling, and synthetic code generation. Each step is described in more detail below.

**Computing Edge Weights**

During this phase of analysis, each node in the graph is visited and transition probabilities are calculated and appended to the edges. Since the program control flow graph is a directed graph, transition probabilities can be computed using the sum of a node's out-edge weights and the weight of each individual edge. The new weights replace the previous counts and the conditional probability function $P(N_n|N_{n-1})$ can be used to evaluate the transition probability for a given node, $N_n$.

**Identifying Child Threads**

While it is straightforward to identify ownership by thread, it is much more difficult to identify which basic block is responsible for a specific thread's management, which is critical when attempting to maintain the characteristics of the original program. In this phase, the algorithm iterates through each node in each statistical flow graph and identify the nodes responsible for spawning a new thread. When a spawn-node is encountered, the address stored as the target function is checked against the address of each basic block in each graph until a match is found. If that node does not yet have an owner, the thread containing the node is recorded as the spawn-target in the parent node. If the thread already has a parent, the search continues until a target is found. When selecting from a pool of available child process that execute the same piece of code, it is impossible to determine when a specific thread is spawned, only that a thread

was spawned with a specific starting address. Because these threads do execute the same piece of code, this does not affect the characteristics of the synthetic workload.

**Wavelet-Based Branch Modeling**

Prior workload synthesis studies [3] use a single global statistic (e.g. taken/not-taken probability) to represent the branch behavior of the original program. To achieve higher accuracy, [36] incorporates transition rates to filter out highly biased branches. To effectively capture workloads' complex branching patterns, the branch of each basic block is profiled and store its dynamic execution (e.g. taken or not-taken) as a bit vector. A trace with length of 32 was found to provide sufficient accuracy to capture branch dynamics of the experimented workloads. Each bit vector is treated as a time series (e.g. 1 stands for taken and 0 represent not-taken) and apply wavelet analysis [14] to extract key patterns of the basic block's branch dynamics. Wavelets can preserve both time and spatial localization. Consequently, the complex branch dynamics can be captured by a few wavelet coefficients. 16 wavelet coefficients are used to capture dynamic branching patterns and apply the K-mean algorithm to classify branching patterns into clusters based on the similarity of their wavelet coefficients. As a result, instead of storing an individual pattern for each branch in synthetic programs, a representative pattern for all branches within the same cluster is used, reducing the overhead of storing each block's branch pattern. Differing with prior work, the branch modeling technique cost-effectively captures complex branch dynamics and is independent of specific microarchitecture implementations.

**Synthetic Benchmark Generation**

The synthetic benchmark is generated by performing a walk of the reduced statistical flow graph. The algorithm used to generate the synthetic multi-threaded

program is described below along with a more in-depth explanation of the code generator (the control flow of code generator is shown in Figure 2-4).

1. Choose the statistical flow graph of the next thread, beginning with thread zero (main thread).
2. Generate a header based on the thread's ID. If the thread ID is zero, emit the program header and information for the main() function. Otherwise, generate a function header to coincide with the thread ID.
3. Begin at the root of the reduced statistical flow graph. If there is no root or the count of the starting node is zero, start with the lowest labeled node that remains.
4. If the node is a tagged as a thread-management point (spawn, destroy, detach etc.), determine which thread is associated with the node's control action, populate the synthetic program with the appropriate assembly-level macro or system call, and proceed to step 6. Otherwise, proceed to step 5.
5. If the node is tagged as a thread-synchronization point (lock, barrier, broadcast, etc.), determine which variable is associated with the node's control action and populate the synthetic program with the appropriate assembly-level macro. Otherwise, proceed to step 6.
6. Pass the node contents to the code generator – instead of generating artificial code based solely on the characteristics of a node, the code generator replicates the original opcode and inserts operands derived from the original operands and the average dependency distance for the instruction. Code is inserted into the synthetic program by prefixing the instructions with the 'asm volatile' label. The

volatile directive prevents the compiler from reordering or optimizing the instructions.

7. Decrement the node instance in the statistical flow graph.
8. A cumulative distribution function, derived from the edge probabilities, is used to determine the next basic block to insert into the synthetic program. If the node has no out-edges and there are still nodes remaining in the graph with instance counts greater than zero, return to step 3. If all of the nodes have been exhausted, return to step 1. Otherwise, using the next basic block, return to step 4.

The functional part of the code generator is broken into five potential phases, outlined in Figure 2-4. If the target instruction is not a branch operation and has no memory operands, then no modification is necessary. If the instruction is a branch, the basic block's cluster ID is used to select the corresponding branch pattern bit vector. Two additional operations are then appended to the basic block to choose the branch target. All taken branch targets are the next-next-basic block while not-taken branches are the next basic block. If the operation accesses memory, the size of the operand and the opcode type are checked to determine the appropriate memory type. A uniform random variable is used to choose the next stride from the histogram. Once the opcode and operands have been determined, the instruction is populated with the corresponding C-style variables and the instruction is written out.

## Evaluation

In this section, the efficiency and accuracy of using synthetic multi-threaded workloads for multi-core performance evaluation is examined. In addition, various

workload and architecture characteristics are compared and contrasted between the synthetic and original multi-threaded benchmarks.

**Experimental Setup**

While the majority of research in workload synthesis and statistical modeling is performed in a simulation environment, the accuracy and efficiency of the proposed techniques were tested across three read-world hardware platforms. A summary of the system configurations for the test platforms are listed in Table 2-1. The evaluations are limited to Intel processor technology in the evaluations due to compatibility with Intel's VTune performance analyzer but the chosen platforms represent three generations of multi-threaded/multi-core hardware. Threads share both pipeline and caches on the Hyper-threading machine. On the Dual Core Pentium D machine, threads run on two separate cores, which only share the front-side bus. The Core 2 Quad machine has four homogeneous cores with an L2 cache shared between every two cores. The Hyper-Threading machine and the Pentium D are similar in that they are based on the same microarchitecture but the Core 2 machine is based on a completely new microarchitecture. A summary of the microarchitecture characteristics for each machine is shown in Table 2-2. These three machines are referenced as HT, Dual, and Quad in this paper.

In this study, nine SPLASH-2 benchmarks [83]: Barnes-Hut (16k Bodies), Cholesky (TK29.0), FFT (220 data points), LU (1024x1024 Matrix), Ocean-Contiguous (258x258 Ocean Body), Ocean-Noncontiguous (258x258 Ocean Body), Water-Spatial (2197 Molecules), Radix (3M keys, 1024 radix) and Volrend (head-scaleddown4) were used. Workload performance and execution characteristics were measured using Intel's VTune analyzer [23]. Since multi-threaded workloads exhibit non-deterministic runtime

behavior, each workload (both original and synthetic versions) was measured using multiple runs and reported average statistics.

**Accuracy**

To evaluate the accuracy of the proposed methodologies, the relative cross-platform speedup obtained from the synthetic benchmarks is gathered and compared with that reported using the original workloads. Note that the raw CPI is a less suitable metric in these evaluations for several reasons, the most important of which is a) the dynamic instruction count can change from run to run and b) the systems do not have a common cycle time. Because multi-threaded programs are used in these evaluations, timing variations can influence the thread interleaving and thus the execution path of the program. This is important because VTune performs sampling during sleep/idle time, spin locks, and other periods where the thread may not be doing useful work. If the synthetic derivation of a program is truly representative of the program from which it is derived, it should exhibit the same relative runtime increases/decreases when it is run on the different machines.

Tables 2-3 compares cross-platform speedup measured using both original and synthetic workloads with four threads. The cross-platform speedup is calculated using the formula: $Speedup(\frac{Quad}{Dual})_{Original} = \frac{ExecutionTime(Dual)_{Original}}{ExecutionTime(Quad)_{Original}}$

In addition, the average absolute errors are computed using an individual workload to measure of all cross-platform speedup (e.g. cross-platform error), and using all benchmarks to measure the speedup of two given platforms (e.g. cross-benchmark error). As can be seen, the maximum error introduced by the synthetic is 14.4%. Overall, the synthetic version of the studied SPLASH-2 benchmarks results in a cross-

platform error ranging from 3.8% to 9.8% and a cross-benchmark error with a margin of error between 6.5% and 7.9%. This suggests that the synthesized benchmarks can be used to accurately evaluate various design alternatives during multi-core design space exploration.

**Efficiency**

To evaluate the effectiveness of applying synthetic multi-threaded workloads to multi-core performance evaluation, the execution runtime of the synthetic programs are compared with that of the original applications. The results are presented as runtime reduction ratio in Table 2-4. In general, more than an order of magnitude decrease in execution time is observed. Because the number of basic blocks emitted during synthesis is different for each program, the synthetic program generated for LU is larger than those generated for the other benchmarks, with respect to the original application, resulting in a higher fraction of runtime. Two of the largest programs, in terms of dynamic instruction counts, are Volrend and Water-SP and the synthetic programs generated for these two applications have two of the shortest runtimes. The technique is expected to easily scale with large contemporary multi-threaded workloads and to produce synthetic programs with several orders of magnitude difference in runtime.

**Workload Characteristics**

The inherent workload characteristics are compared, including dynamic instruction distribution and mix, between original and synthetic workloads. The instruction count distribution between the synthetic and original programs correspond very well, with little deviation – less than 8% on average. This implies that the techniques are capable of capturing thread activities and appropriately scaling down individual thread run time. Figure 2-5 illustrates instruction mix between the original and the synthetic FFT

benchmarks. As can be seen, the instruction mix in the synthetic program and the original program is similar. The differences are because the code generator must swap some instructions for others (e.g. cmov → mov) because no attempt is made to preserve values in the synthetic workload.

**Microarchitecture Characteristics**

A variety of microarchitecture performance characteristics are examined using 4-thread synthetic workloads. Each metric is compared with those of the original program. Figure 2-6 shows a comparison of CPI, L1 data cache and L2 cache hit rates, and branch prediction accuracy on the Pentium D system. Microarchitecture characteristics are analyzed on the HT and Core 2 Quad machines and their error trends are similar. The maximum CPI discrepancy is 12% (Ocean-cont). The wavelet-based branch model accurately and cost-effectively captures branch dynamic behavior, resulting in an error margin less than 4%. Converging memory behavior between the synthetic and the original is more challenging, the thread-aware memory reference model overestimates L1 data cache performance on workloads Ocean-Cont, Ocean-Non, Barnes, LU, and FFT. The estimated L2 cache performance shows less discrepancy. This is because the original SPLASH-2 workload datasets easily fit into the processor L2 caches.

All references to the L2 cache are broken down based on the states of a cache block. The results on the Core 2 Quad platform are shown in Figure2- 7. A MESI based coherency protocol is used by the Core 2 Quad processors to maintain the data consistency. The coherence protocol transitions the state of each L2 cache line between Modified (M), Exclusive (E), Shared (S), and Invalid (I) to reflect the current cache line status among the four cores. The MESI-based L2 access breakdown reveals the data sharing patterns between threads. If a synthetic workload faithfully captures the

data sharing characteristics of its original counterpart, they both will exhibit a similar breakdown of these events. The thread-aware memory reference model that captures both private and shared data access patterns as well as the read and write ratio of each access pattern is responsible for these similarities. The results shown in Figure 2-7 suggest that both the original and the synthetic workloads stress cache coherency hardware similarly and will generate similar coherence traffic among the multiple cores.

**Data Sharing and Thread Interaction**

The advanced multi-core performance counters provided by the Core 2 Quad processors are used to analyze the impact of thread interaction on both the synthetic and original workloads. To be more specific, VTune's modified data sharing ratio, locked operations impact, and data snoop ratio are examined. The modified data sharing ratio measures the frequency of data sharing one two or more threads modify the data in one cache line. The locked operations impact is a measure of the penalty due to operations using the IA-32 LOCK prefix. The data snoop ratio is a measure of how often a cache is snooped by an adjacent or external processing element. The results of 4-thread workloads, shown in Tables 2-4 and 2-5, indicate that the synthetic significantly scales down the runtime while still faithfully preserving thread interaction.

**Limitations**

In this research, real hardware platforms are used since the non-deterministic execution characteristics of the multi-threaded workloads cannot be captured using current cycle-accurate simulators. However, the use of real hardware limits the number of configurations and the scope of the design space. In future work, additional studies will be performed using simulators and compare the results with those obtained using real hardware. The framework is built around the Pthread libraries but can be extended

to use OpenMP, UPC, MPI, or a combination of programming models. The Pthread model makes the SPLASH-2 suite the natural place to begin tests plans are underway to include commercial and server multi-threaded workloads.

## Related Work

SimPoint [73] and SMARTS [84] apply machine learning and statistical sampling to reduce the average number of instructions required for detailed, cycle-accurate simulation of each benchmark. SimPoint and SMARTS have been shown to be quite successful for single threaded applications. On-going efforts [5] suggest that it becomes more challenging to apply these mechanisms to multi-threading/multi-core scenarios since sampling can result in simplifications that can miss non-deterministic executions, complex interactions between the multiple threads and the operating system, and parallelism among the multiple cores.

Recent proposals have used statistical simulation [16][23][56][55][17][18][22][38][57] to reduce architecture simulation time. Statistical simulation measures characteristics during normal program execution, generates a synthetic trace with those characteristics, and then simulate the synthetic trace. The statistically generated synthetic trace is orders of magnitude smaller than the original program sequence and results in significantly faster simulation. For single threaded benchmarks, Nussbaum & Smith and Eeckhout et al. both showed that statistical simulation can quickly converge (within 10k to 100k cycles) to a performance estimate typically within 5% error when compared to detailed simulation [16][55]. Nussbaum and Smith built the first statistical multiprocessor model [56] and reported errors less than 15%, on average, for the SPLASH-2 benchmarks. Their approach incorporated barrier, lock, and critical section distributions derived from their source programs. Their cache

and branch models are limited to the cache and predictor configurations for which the statistics were collected. More recently, [22] used statistical simulation to model multi-programmed workloads in a CMP in an architecturally independent fashion. Their simulator is able to model the shared cache structure and the program's time-varying behavior. In this work, workload characterization techniques are used to capture fine-grained, microarchitecture impendent thread interaction, memory accesses, and branch behavior. The framework is capable of generating re-compilable and portable miniature benchmarks that execute on real hardware and target the most complex commercially available x86 ISA. In addition, both the accuracy and efficiency of synthetic multi-threaded workloads across three real-world multi-threaded/multi-core processors are reported. This paper presents the first work to accurately and automatically synthesize multi-threaded workloads. [12] proposed segmenting the simulator into separate software and hardware components with the hardware component managed by a FPGA. These simulators are capable of executing 1M to 100M cycles per second. The synthetic workloads can be applied to a FPGA-based simulation accelerator to further reduce the simulation time.

## Summary

Multi-core design evaluation is extremely time-consuming because of the number of elements involved in any thorough design study. This exploration is likely to become even more time consuming as the number of cores per die increases. The workload synthesis methods described in this paper for multi-threaded programs attempts to address this problem. Employing techniques from statistical simulation, synchronized statistical flow graphs for multi-threaded programs are generated. These graphs contain not only the individual thread attributes but also the inter-thread synchronization and

sharing characteristics. Using the novel thread-aware memory reference models and the wavelet-based dynamic branch models, the tool accurately captures and cost-effectively preserves memory locality and branch behavior of the original multi-threaded workloads. Combined with memory and branch models, the synchronized statistical flow graphs can be used to automatically generate a multi-threaded synthetic workload comprised of the dynamic execution features of the original program. The synthetic program is emitted as a series of low-level statements embedded in C. When compiled, the synthetic program maintains the dynamic characteristics of the original program but with significantly reduced runtime. Because the synthetic code can be compiled into a new binary, it can be executed on a variety of platforms. The framework is modular and is expected to extendable to encompass a variety of threading languages and ISAs.

Table 2-1. Configuration of the experimental platforms

| Parameter | Platform A | Platform B | Platform C |
|---|---|---|---|
| Processor | Pentium 4 | Pentium D | Core 2 Quad |
| Memory | 1024MB DDR400 | 4096MB DDR2-4200 | 4096MB DDR2-4200 |
| Storage | 80GB SATA | 160GB SATA | 180GB SATA |
| Operating System | SuSE 10.01 | SuSE 10.01 | SuSE 10.2 |

Table 2-2. Microarchitecture characteristics for the experimental platforms

| Parameter | Pentium 4 | Pentium D | Core 2 Quad |
|---|---|---|---|
| PEs | 1 Physical/2 Virtual | 2 Physical | 4 Physical |
| Tech | 130nm | 90nm | 65nm |
| Clock Speed | 2.4GHz | 2.8GHz | 2.4GHz |
| FSB | 400MHz | 800MHz | 1066MHz |
| Trace Cache | 12k uOps | 12k uOps | -- |
| L1I Cache | -- | -- | 4x32kB 8-way |
| L1D Cache | 1x8kB 4-way | 2x16kB 8-way | 4x32kB 8-way |
| L2 Cache | 1x512kB 8-way | 2x1MB 8-way | 2x4MB 16-way |
| ROB Size | 123 | 126 | 96 |
| IUs | ALU:3 AGU:2 | ALU:3 AGU:2 | ALU:3 AGU:2 |
| FPUs | 2 | 2 | 2 |

Table 2-3. Cross platform speedup. The cross-platform speedup is calculated using the workload's execution time on two out of the three platforms.

| | | *Barnes* | *Cholesky* | *FFT* | *LU* | *Ocean-C* | *Ocean-NC* | *Water-SP* | *Radix* | *Volrend* | Cross Bench-mark Error |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Quad /Dual | Original | 2.26 | 1.75 | 1.26 | 1.67 | 1.23 | 1.63 | 1.73 | 1.84 | 2.73 | 7.9% |
| | Synthetic (Error) | 2.04 -9.8%) | 1.92 (9.7%) | 1.30 (3.3%) | 1.53 (-8.6%) | 1.1 -10.3%) | 1.53 (-6.1%) | 1.63 -5.6%) | 1.74 (-5.6%) | 3.05 (11.7%) | |
| Quad /HT | Original | 2.87 | 1.8 | 1.96 | 3.03 | 2.8 | 3.45 | 2.93 | 2.28 | 3.92 | 6.5% |
| | Synthetic (Error) | 2.87 (0%) | 1.98 (10%) | 2.12 (8.5%) | 2.64 (-12.9%) | 2.84 (1.3%) | 2.95 (-14.4%) | 2.93 (0%) | 2.41 (5.5%) | 4.14 (5.6%) | |
| Dual /HT | Original | 1.27 | 1.02 | 1.55 | 1.82 | 2.28 | 2.12 | 1.7 | 1.24 | 1.44 | 7.3% |
| | Synthetic (Error) | 1.41 (11%) | 1.03 (0.3%) | 1.63 (5%) | 1.73 (-4.7%) | 2.57 (12.9%) | 1.93 (-8.8%) | 1.8 (5.7%) | 1.38 (11.8%) | 1.36 (-5.6%) | |
| Cross Platform Error | | 6.9% | 6.7% | 5.6% | 8.7% | 8.2% | 9.8% | 3.8% | 7.6% | 7.6% | |

Table 2-4. A comparison of runtime reduction ratio between synthetic and original multi-threaded workloads

| | *Barnes* | *Cholesky* | *FFT* | *LU* | *Ocean-C* | *Ocean-NC* | *Water-SP* | *Radix* | *Volrend* |
|---|---|---|---|---|---|---|---|---|---|
| HT | 290 | 145 | 15 | 9 | 21 | 15 | 335 | 12 | 357 |
| Dual | 261 | 144 | 14 | 9 | 19 | 17 | 316 | 11 | 378 |
| Quad | 236 | 158 | 14 | 8 | 17 | 16 | 298 | 10 | 422 |

Table 2-5. Thread interaction comparison

| | | *Barnes* | *Cholesky* | *FFT* | *LU* | *Ocean-C* | *Ocean-N* | *Water-SP* | *Radix* | *Volrend* |
|---|---|---|---|---|---|---|---|---|---|---|
| **Locked Operations Impact** | Original | 0.2% | 1.3% | 0.8% | 0.3% | 2.2% | 2.6% | 0.1% | 0.6% | 2.3% |
| | Synthetic Error | 3.5% | 17.6% | -3.2% | 6.6% | -3.2% | 9.2% | -11.4% | -2.7% | 11.7% |
| **Modified Data Sharing Ratio per 1k Instructions** | Original | 0.2 | 0.3 | 0.2 | 0.1 | 0.0 | 3.1 | 0.2 | 0.2 | 0.2 |
| | Synthetic Error | -3.5% | 11.6% | 7.7% | -10% | 1% | -9.2% | 4.4% | 2.6% | 5.6% |
| **Data Snoop Ratio per 1k Instructions** | Original | 21 | 14 | 46 | 9 | 55 | 75 | 3 | 23 | 3 |
| | Synthetic Error | -7% | -4.8% | -7.7% | 13.2% | 3.5% | 6.8% | -3.4% | -1.6% | -5.6% |

```
#include <stdlib.h>
#include <pthread.h>

void *myFunction(void *ptr);

pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;
size_t myUnsigned = 7;

int main(int argc, char *argv[])
{
   pthread_t threadA, threadB;

   pthread_create(&threadA, NULL, &myFunction, NULL);
   pthread_create(&threadB, NULL, &myFunction, NULL);

   pthread_join(threadA, NULL);
   pthread_join(threadB, NULL);
   return 0;
}

void *myFunction (void *ptr)
{
   pthread_mutex_lock(&myMutex);
   usleep(2);
   myUnsigned =  myUnsigned + 1;
   myUnsigned =  myUnsigned * 3;
   myUnsigned =  myUnsigned + 10;
   pthread_mutex_unlock(&myMutex);
}
```

Figure 2-1. A sample multithreaded program

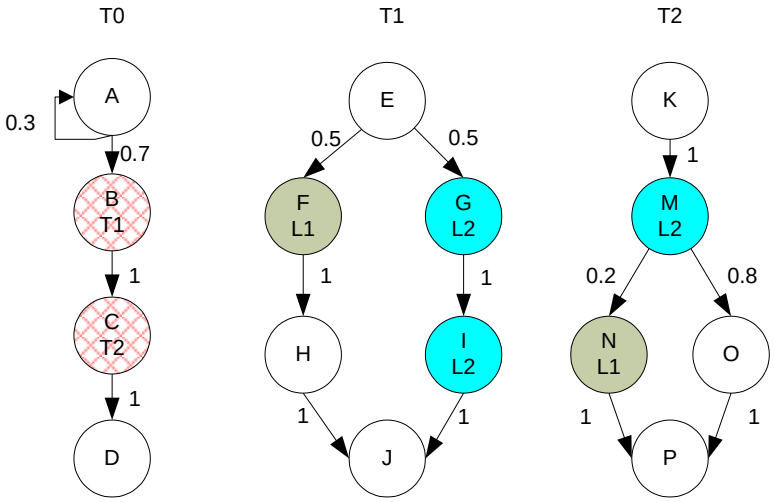


Figure 2-2. Sample SSFG. Edges are annotated to show transition probabilities and nodes are annotated to show control points (B and C in T0) and critical sections (F, G and I inT1 and N and M in T2) which are protected by locks L1 and L2

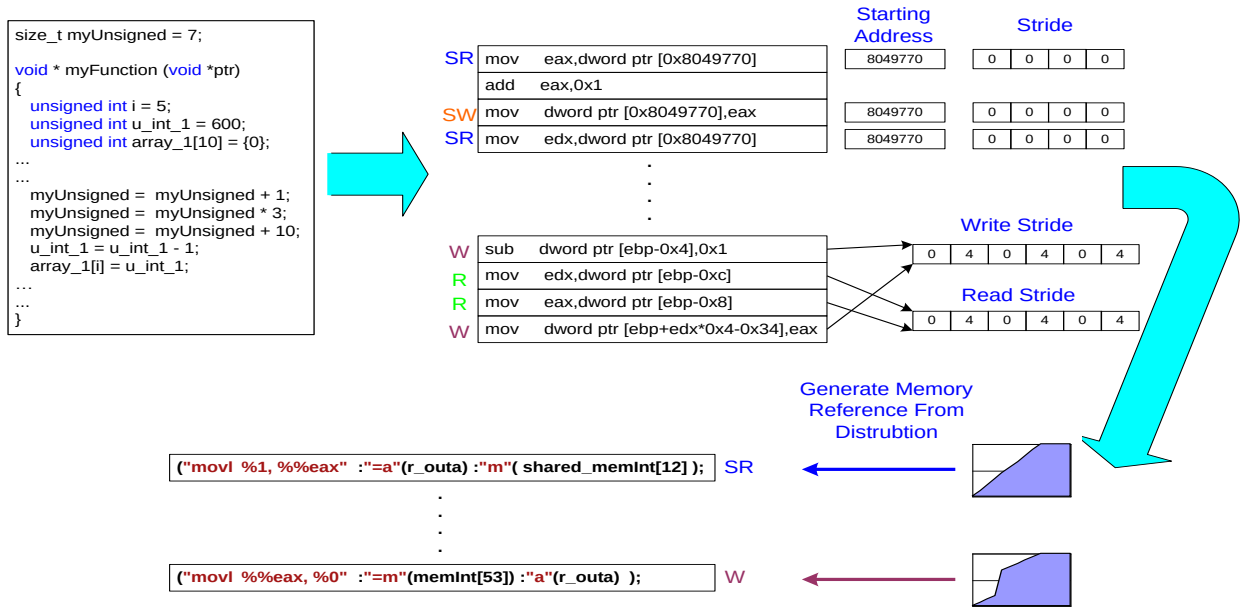


Figure 2-3. Thread-aware memory reference model

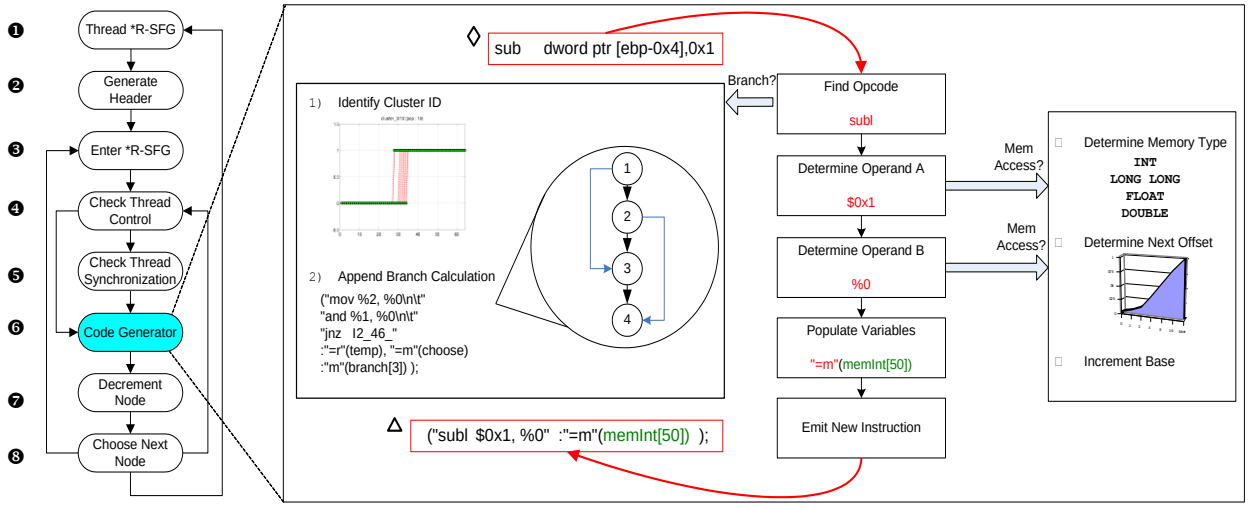


Figure 2-4. Control flow in code generator – *: Reduced SFG, ◊: Instruction from Pin, Δ: Synthesized instruction

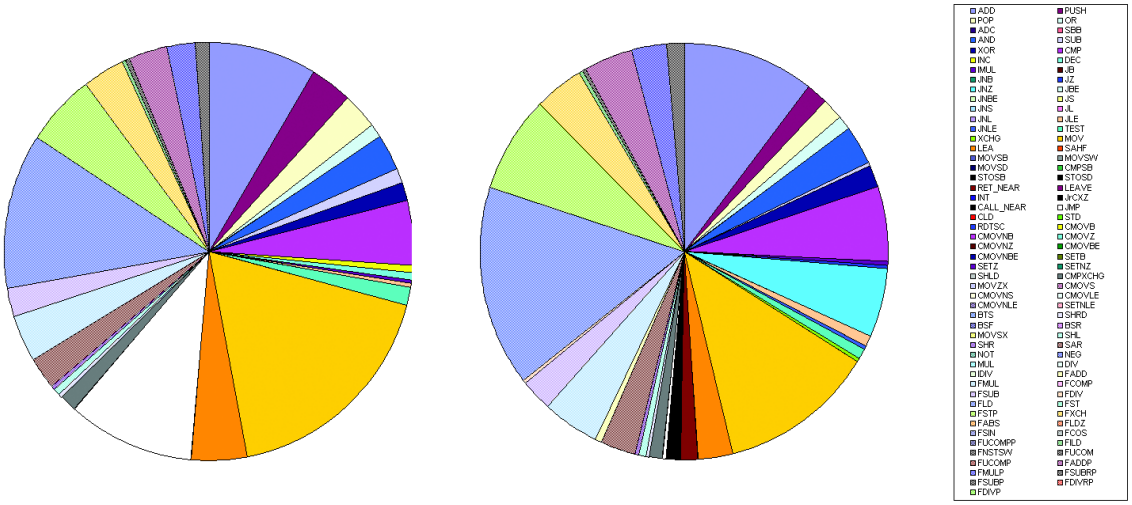


Figure 2-5. A comparison of instruction mix between synthetic (left) and original (right) FFT

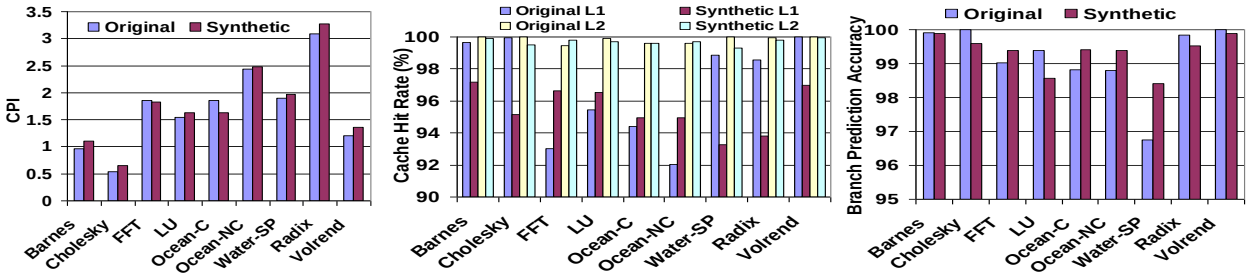


Figure 2-6. A comparison of CPI, cache hit rates, and branch prediction accuracy of the synthetic and original workloads

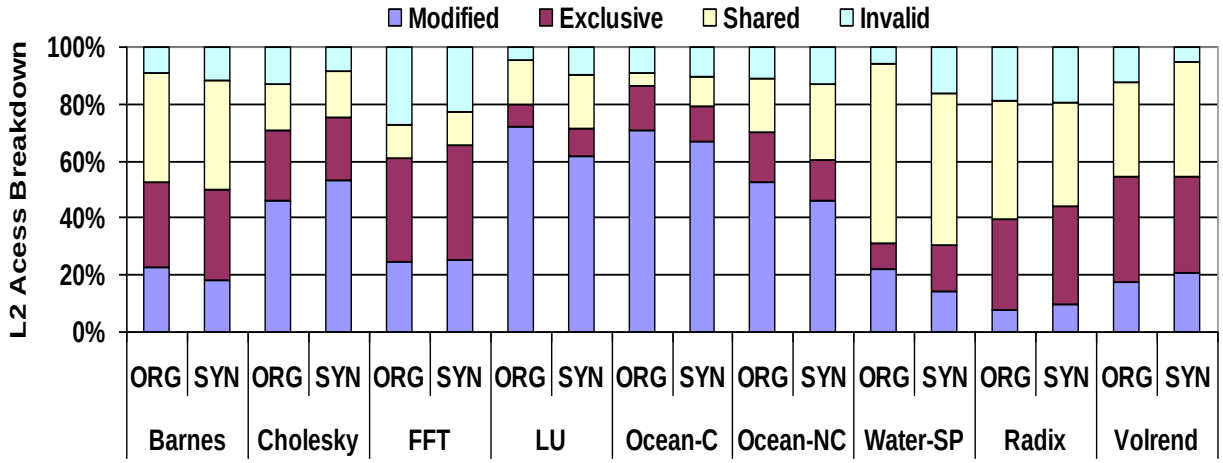


Figure 2-7. L2 Access breakdown by MESI states

# CHAPTER 3
# TRANSPLANT: A PARAMETERIZED METHODOLOGY FOR GENERATING TRANSACTIONAL MEMORY WORKLOADS

## Background and Motivation

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- [16] or multithreaded [56][29] 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.

## 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 this work.

**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[35], BioParallel [34], ALPBench[44], MineBench [54], SPEComp [76], SPLASH-2 [83], and PARSEC [4]. 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.

**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?

**Benchmark Redundancy**

Previous authors have shown that many programs within a benchmark suite exhibit tremendous amounts of redundancy [64][19][17][37]. 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 [4]. 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 3-2). An overview of the mathematical processes involved in this evaluation can be found in Section 3-4.2.

Figure 3-1 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 3-9) 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 stress all of the elements in its design and that new programs may be needed.

**Benchmark Synthesis**

Statistical simulation [56] and workload synthesis [16] 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 [56][29]. Although this previous work does produce small fast-

running programs, it differs from this tool in that the proposed methodology does not use any program as a starting point. The 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. [37] 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.

## TransPlant

In the following section the TransPlant model for generating transactional workloads is introduced. Descriptions are provided on how it both differs from and expands upon currently available transactional benchmarks. The discussion ends with details concerning its capabilities and on the implementation of the TransPlant framework.

### 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.

**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 3-1 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 the 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.

**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 3-2.

**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.

**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, an attempt is made 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.

**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, a stride value based on the instruction's privatization, localization, and confliction parameters is assigned. Maps are maintained for 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.

**Code Generation**

SESC [72] is used as the simulation environment, so 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.

## Methodology

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

### Transactional Characteristics

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 3-2 describes the features 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. The 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, the transaction percentage, transaction size, read-/write-set conflict densities, and the read-/write-set sizes of each transaction are recorded. Since many transactional workloads exhibit heterogeneous transactions and different synchronization patterns throughout runtime execution, the goal was 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 only committed and not aborted transactions are considered 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, read conflict density and write conflict density are also included. 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, the entire workload is run to completion and the read/write sets for each transaction are calculated. 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 captures 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 permits categorization of 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, 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) are included. 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 3-2 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.

**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 ($X_1$, $X_2$, …, $X_p$), a principal component $Y_1$ is represented as $Y_1 = a_{11}X_1 + a_{12}X_2 + \ldots + a_{1p}X_p$ , where ($Y_1$, $Y_2$, …, $Y_p$) are the new uncorrelated variables (principal components) and ($a_{11}$, $a_{12}$, …, $a_{1p}$) 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 [63] 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. The 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.

## 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, the transactional characteristics of the new benchmarks are measured and the results are evaluated using principal component analysis and clustering. All benchmarks are run to completion with 8-threads using SuperTrans [60]. SuperTrans is built on SESC [72] and is a cycle accurate, multiple-issue, out of order common chip multiprocessor (CMP) simulator that supports cycle accurate simulation of eager and lazy conflict detection and eager and lazy version management. Table 3-3 presents the microarchitecture configuration that was used for each core in the 8-core CMP simulation.

### 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 3-3 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.

**Workload Comparison**

In this section, the overall program characteristics of the benchmarks generated in Section 3-5.1, *Q1-1-Q4-1*, are compared with those of the SPLASH-2 and STAMP benchmarks. Specifically, the same principal component analysis as above is applied with the addition of the new benchmarks. With the reduced data from PCA, hierarchical clustering is used to group the benchmarks. The transactional performance of the benchmarks is evaluated across two different transaction designs.

**Clustering**

Figure 3- 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 3-5.1. 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 3- 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 3- 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 3-5.2.2 confirm this behavior.

The clustering results in Figure 3- 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 [87], *Q2-1* and *Q3-1* are always included.

**Performance**

In order to validate the abstract characteristics discussed above, this section presents 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 3-4. 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.

**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 [60] 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, the aborted cycles to total cycles to average transaction size ratio is used. Table 3-5 shows the results when *testCase* is compared 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*.

**Benchmark Mimicry**

While being able to create benchmarks based on an arbitrary input is useful for testing and debugging, it is important that the tool be able to replicate the behavior of existing benchmarks. In this section, PCA and clustering are used to show how the tool

can use a trace to generate a synthetic benchmark that maintains the high-level program characteristics of the SPLASH and STAMP benchmarks.

Figure 3-7 shows the plot of the first two principle components of the STAMP and SPLASH benchmarks using the inputs from Table 3-2. These two factors comprise 48.9% of the total variance. Figure 3- 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 3-4.1. 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 3- 9 shows the hierarchical clustering for the original applications and Figure 3- 10 shows the clustering for the synthetic representation. While the amalgamation schedule is slightly off, the overall representation is almost exact.

Finally, Figure 3-11 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 direct control in the synthetic benchmark creation, the contentious behavior is a result of the workload's interaction with the transactional model. From Table 3-4, 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 the synthetic benchmarks are able to preserve the architecture independent workload characteristics of the original benchmarks, Figure 3-11 clearly shows that the synthetic benchmarks also preserve the alignment and fine-grain behavior of the original benchmarks.

## Summary

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. The goal was 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, TransPlant is shown to be 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, it is 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.

Table 3-1. 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) |

Table 3-2. 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. |

Table 3-3. 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 |

Table 3-4. 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* | EE | 70533 | 1554 | 2.33 | 6.71 | 6.53 | 1.07 | 204.09 |
| 16K particles | LL | 69336 | 362 | 6.880302 | 6.71 | 6.53 | 1.07 | 204.10 |
| *fmm* | EE | 45256 | 3 | 0.001771 | 13.43 | 7.34 | 1.82 | 175.60 |
| 16K particles | LL | 45302 | 26 | 0.516338 | 13.43 | 7.34 | 1.82 | 175.52 |
| *cholesky* | EE | 15904 | 19 | 0.015719 | 3.13 | 1.95 | 2.01 | 27.18 |
| tk15.O | LL | 15963 | 78 | 0.057466 | 3.12 | 1.95 | 2.01 | 27.16 |
| *ocean-con* | EE | 2161 | 497 | 0.091549 | 3.00 | 0.27 | 12.93 | 10.39 |
| 258x258 | LL | 1800 | 136 | 0.022013 | 3.00 | 0.26 | 13.44 | 10.38 |
| *ocean-non* | EE | 7200 | 5200 | 0.783498 | 3.00 | 0.38 | 9.79 | 13.25 |
| 66x66 | LL | 2778 | 778 | 0.057183 | 3.00 | 0.36 | 10.22 | 13.17 |
| *raytrace* | EE | 141020 | 64279 | 22.43765 | 6.49 | 2.46 | 5.33 | 60.87 |
| Teapot | LL | 307376 | 230635 | 0.260170 | 7.49 | 2.46 | 6.51 | 73.54 |
| *water-nsq* | EE | 10398 | 22 | 0.002693 | 10.87 | 2.97 | 2.66 | 59.26 |
| 512 molecules | LL | 10482 | 106 | 0.654037 | 10.87 | 2.97 | 2.66 | 59.26 |
| *water-sp* | EE | 153 | 0 | 0.000146 | 2.48 | 1.37 | 1.68 | 133.25 |
| 512 molecules | LL | 226 | 73 | 0.003986 | 2.57 | 1.46 | 1.89 | 366.78 |
| *bayes* | EE | 714 | 221 | 65.621712 | 151.65 | 77.62 | 1.95 | 80913.12 |
| 1024 records | LL | 733 | 222 | 0.071399 | 154.78 | 80.63 | 1.91 | 84540.69 |
| *genome* | EE | 6081 | 167 | 1.291080 | 35.78 | 9.62 | 3.71 | 2451.98 |
| g256 s16 n16384 | LL | 6195 | 281 | 1.156334 | 35.76 | 9.63 | 3.71 | 2452.32 |
| *intruder* | EE | 16658 | 5442 | 4.027422 | 14.02 | 8.84 | 1.58 | 494.65 |
| a10 l4 n2038 s1 | LL | 18646 | 7430 | 0.434436 | 13.90 | 8.82 | 1.57 | 494.46 |
| *kmeans* | EE | 6710 | 5 | 0.014471 | 7.31 | 2.74 | 2.66 | 347.04 |
| Random1000_12 | LL | 7075 | 370 | 0.044840 | 7.31 | 2.74 | 2.66 | 347.04 |
| *labyrinth* | EE | 382 | 174 | 323.617009 | 287.10 | 199.29 | 1.44 | 387340.10 |
| 512 molecules | LL | 694 | 486 | 0.048621 | 276.74 | 199.18 | 1.38 | 346683.35 |
| *ssca2* | EE | 6758 | 32 | 0.013136 | 6.19 | 3.04 | 2.03 | 35.13 |
| s11 i1.0 u1.0 l3 p3 | LL | 6941 | 45 | 0.075905 | 6.17 | 3.04 | 2.02 | 35.17 |
| *vacation* | EE | 4096 | 0 | 0.036366 | 75.29 | 16.57 | 4.54 | 4558.53 |
| 4096 tasks | LL | 4107 | 11 | 0.051667 | 75.29 | 16.57 | 4.54 | 4558.52 |
| *yada* | EE | 6573 | 1265 | 123.165861 | 55.85 | 26.84 | 2.08 | 16079.54 |
| a20 633.2 | 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 |

Table 3-4. Continued

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *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**

Table 3-5. 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 |

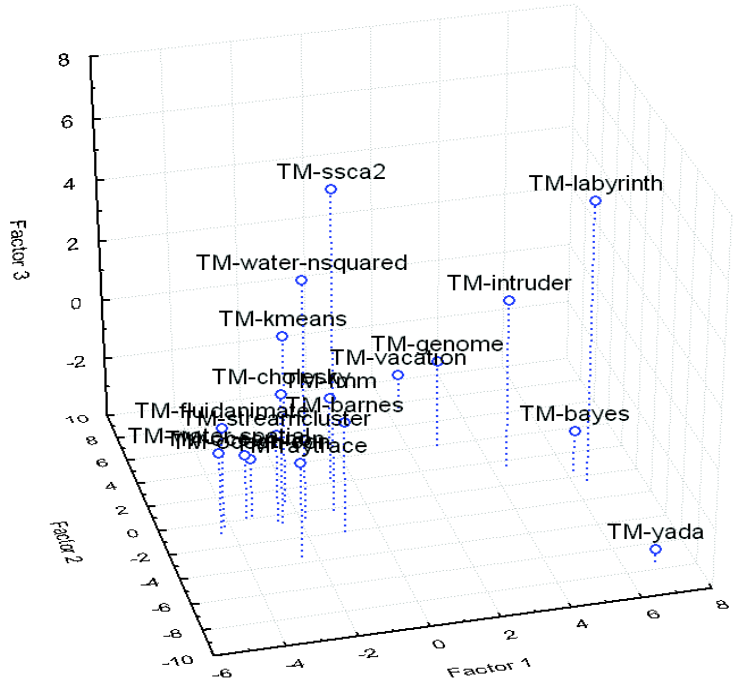


Figure 3-1. PC Plot of STAMP & SPLASH-2

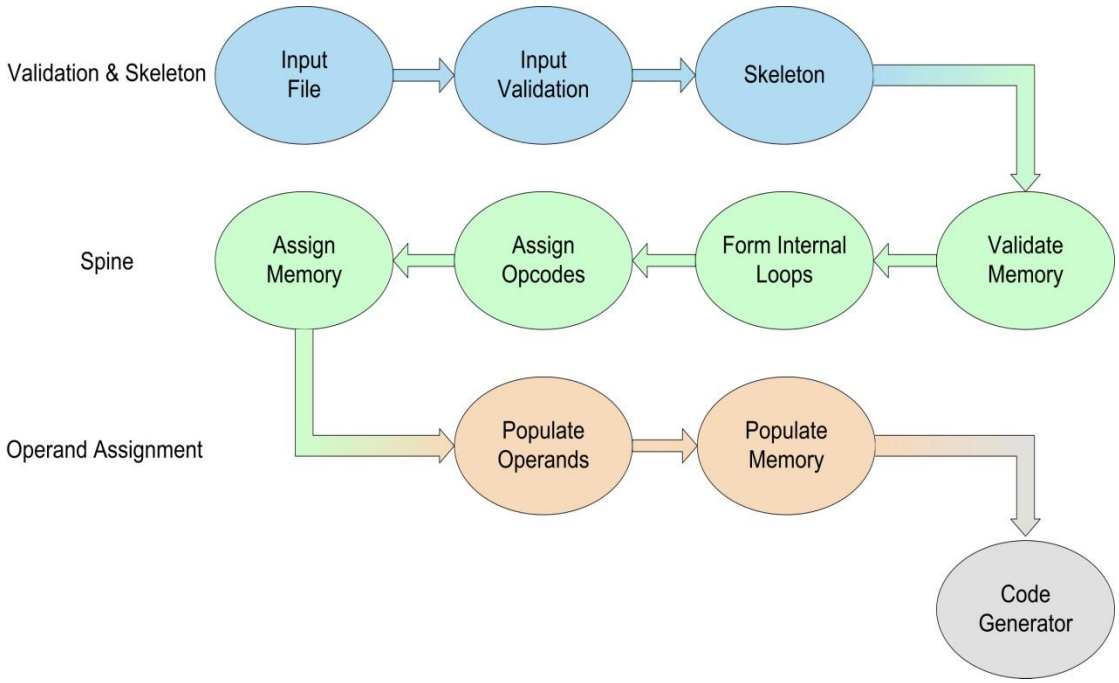


Figure 3-2. High-level Representation of TransPlant

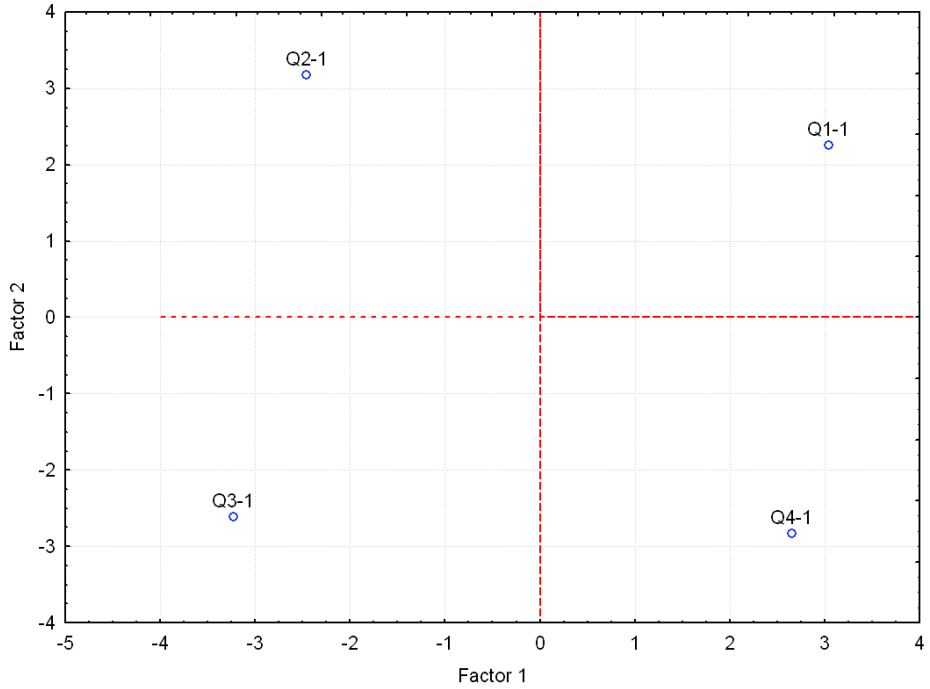


Figure 3-3. PC1-PC2 Plot of Synthetic Programs

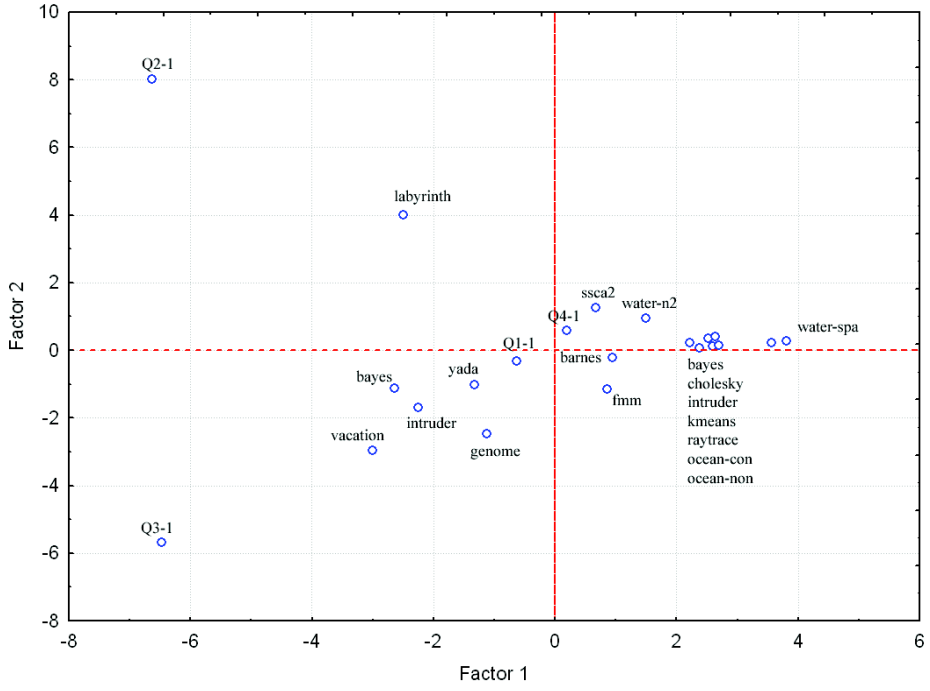


Figure 3-4. PC1-PC2 Plot of Unified PCA

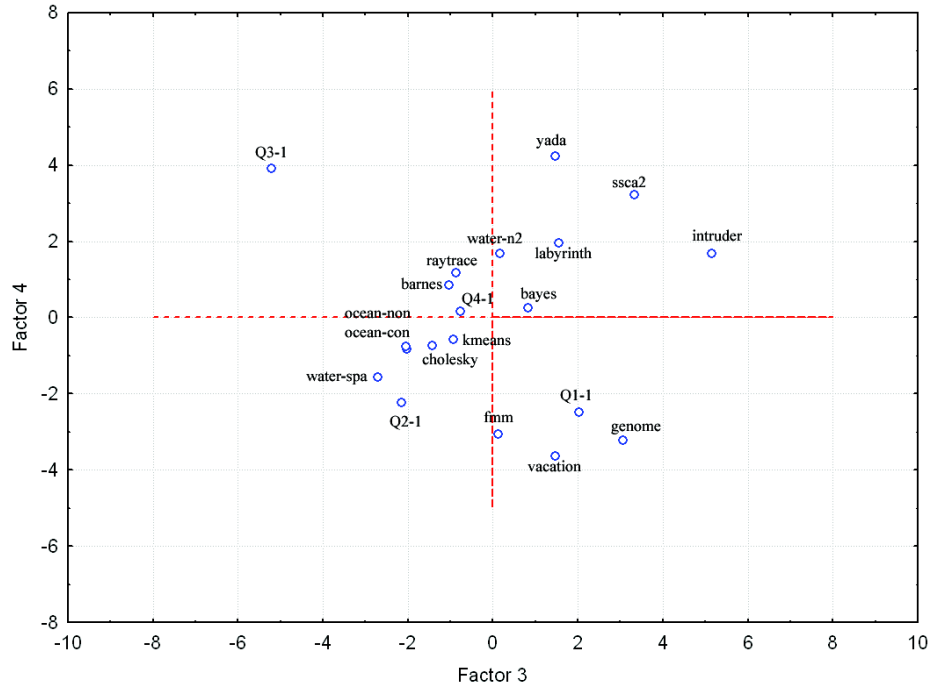


Figure 3-5. PC3-PC4 Plot of Unified PCA

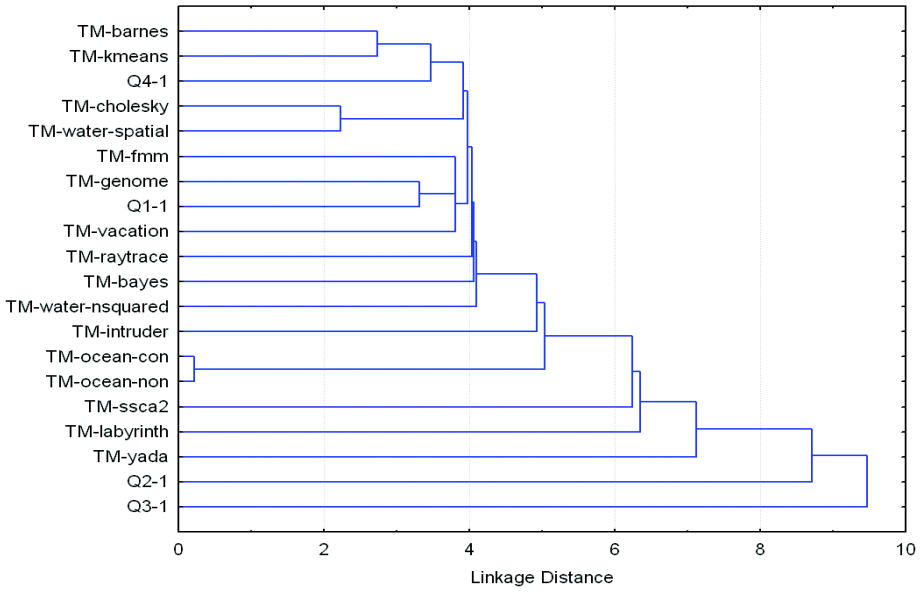


Figure 3-6. Dendrogram (Unified)

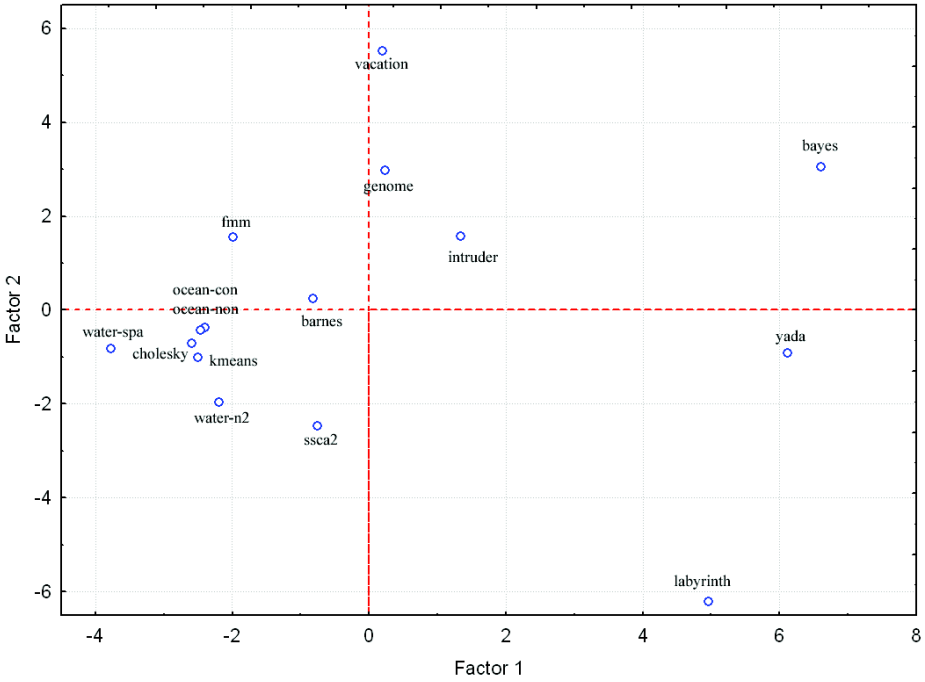


Figure 3-7. PC1-PC2 Plot of Original Applications

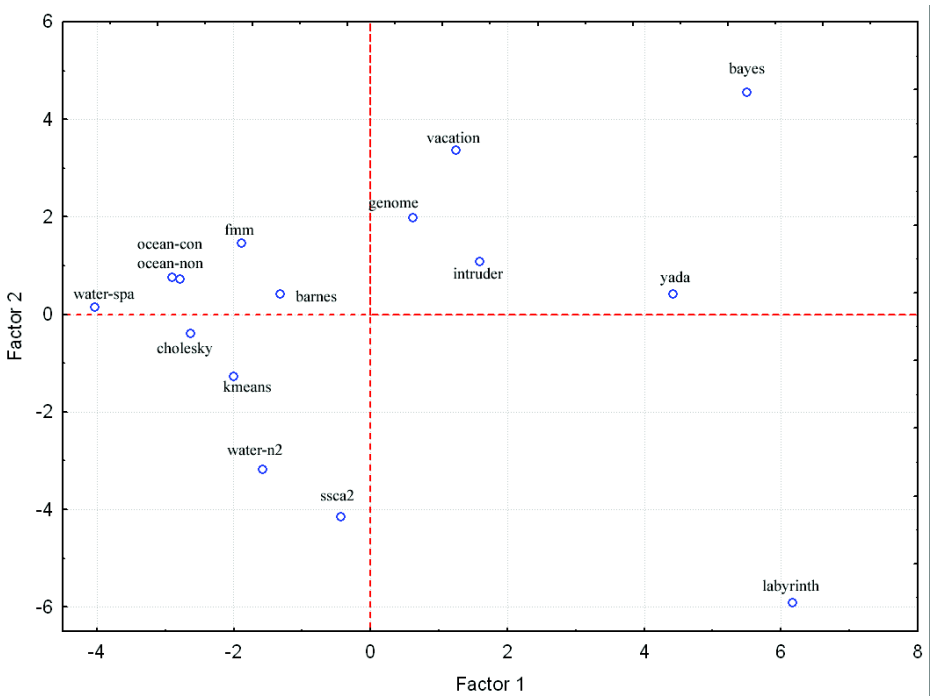


Figure 3-8. PC1-PC2 Plot of Synthetic Applications

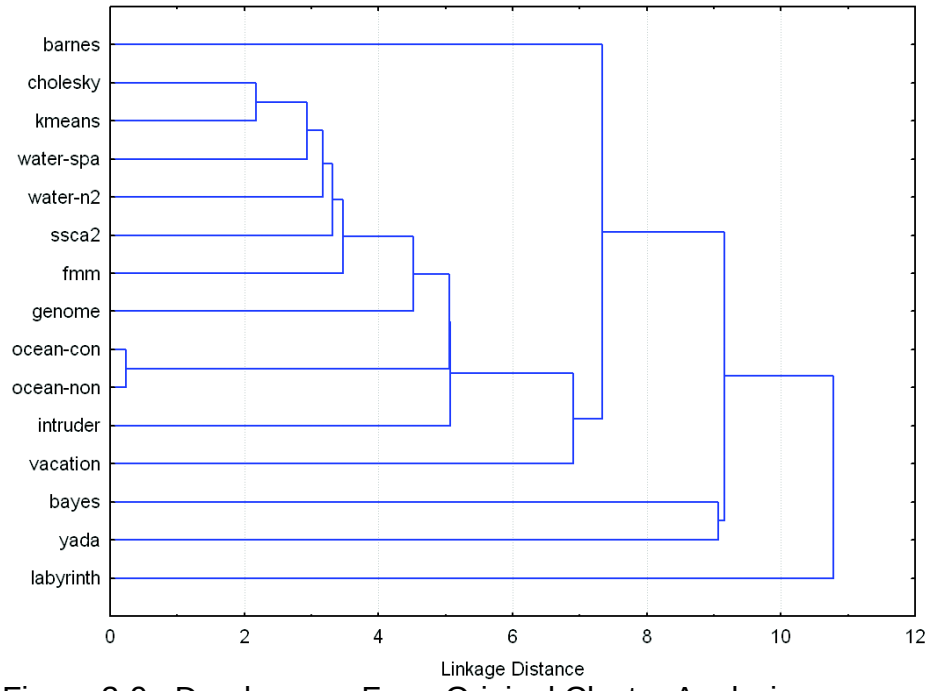


Figure 3-9. Dendrogram From Original Cluster Analysis

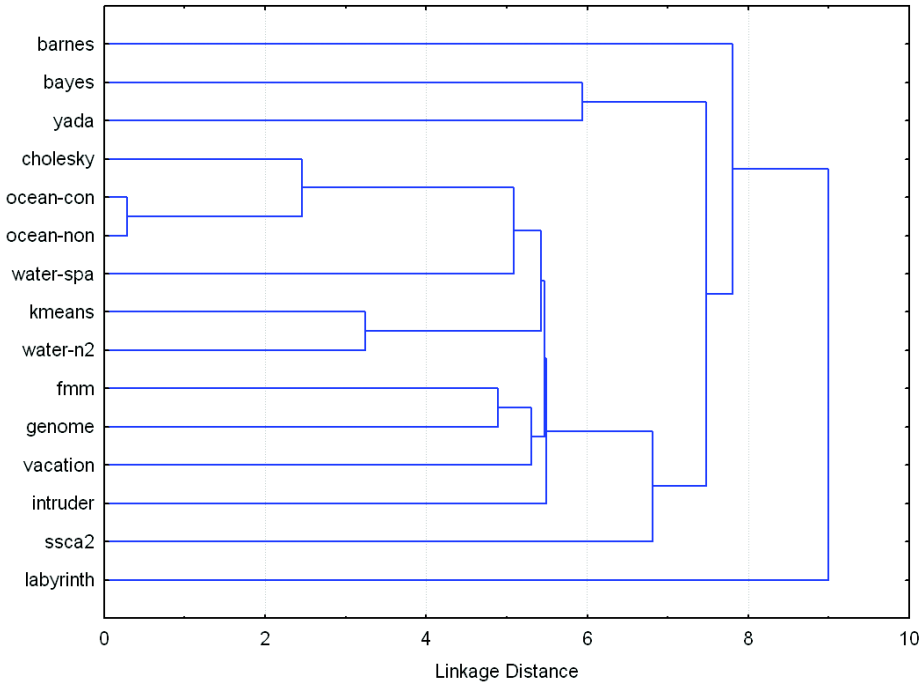


Figure 3-10. Dendrogram From Synthetic Cluster Analysis

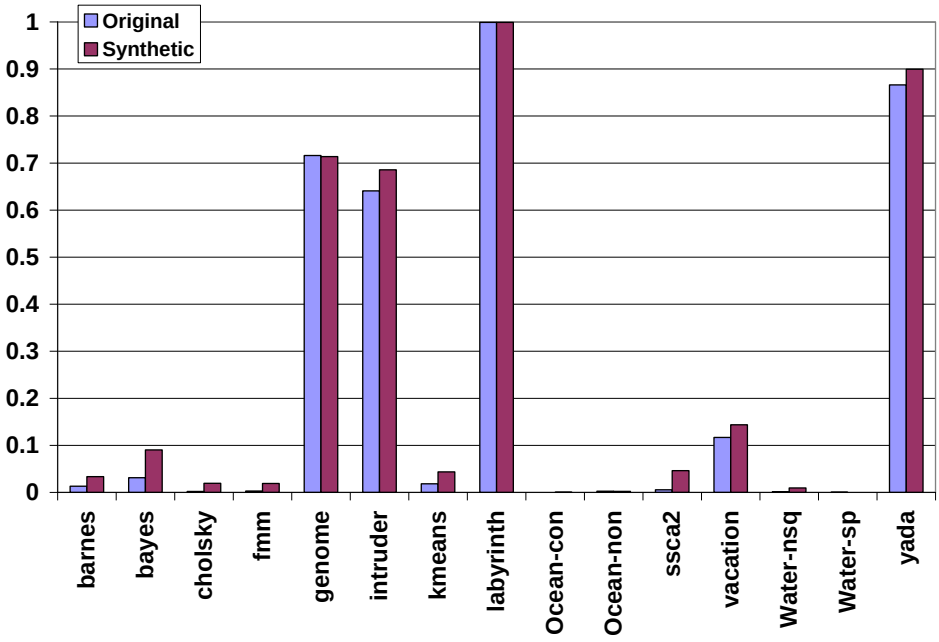


Figure 3-11. Transactional Cycles – Total Cycles

# CHAPTER 4
# POWER-PERFORMANCE IMPLICATIONS FOR HARDWARE TRANSACTIONAL MEMORY

## Background and Motivation

Chip multiprocessors (CMPs) are redefining how architects approach power management, which is a growing concern in all areas of the market from embedded systems to data centers. CMPs still suffer from the same heat removal problems as previous generations of processors and, driven by the ever-increasing number of transistors, power management is now the primary design issue across most application segments [71]. While CMPs offer better energy efficiency than uniprocessors[42], because they share some resources such as cache, memory buses, and memory banks, the power distribution changes as the number of processing elements (PEs) increases [20]. This is because each PE affects the power and performance of all collocated PEs.

As CMPs penetrate more of the marketplace, programmers will need to begin changing the way they write code to be able to take advantage of the resources available on CMPs and the ever-increasing number of processing elements. However, exploiting the available data and task parallelism in a program is often a challenging and time-consuming process, requiring significant time investments to extract performance and guarantee correctness. Transactional memory [26] has been proposed as a programming technique to replace locks, shifting some of the burden of synchronization from the programmer to the architecture, in effect providing an abstraction of the implementation.

There have been numerous design proposals for hardware transactional memory systems ) [26] [24] [39], software transactional memory systems [15] [25] [65], and

hybrid transactional memory systems [51] [66] [74] but they can all be described by two primary design points:  conflict detection and version management. Conflict detection defines when conflicts are detected and version management defines where new and old values within a transaction are stored. Both use the same basic nomenclature and can be either eager or lazy. With eager conflict detection, addresses are checked for conflicts on each read and write within the transaction whereas lazy checks addresses when a transaction attempts to commit. Eager version management writes new values in place and copies old values elsewhere; lazy does the opposite, leaving old values in place and writing new values elsewhere. While transactional memory was conceived of as a means to shift the programming burden, hardware implementations have ancillary benefits such as increased performance over locks and potential energy savings [53]. However, to date no complete work has been undertaken to compare the energy and performance tradeoffs for hardware implementations.

This work compares the power consumption of hardware transactional memory using eager and lazy versioning and conflict detection schemes with a lock-based system. Using benchmarks from SPLASH-2 [83] and STAMP [50], chosen since they are the most commonly used benchmarks for multi-core and transactional memory evaluation, it is shown that, on average, hardware transactional memory consumes less power than an equivalent lock-based program. However, when the power and performance are considered jointly, the lock-based programs outperform the transactional memory systems. These results are directly related to the contention within the benchmarks and how the transactional models handle conflicts. Synthetic workloads are used to reinforce the conclusions drawn from the real benchmarks. Using

these benchmarks, it is shown that the power-performance of the eager conflict detection schemes can vary wildly when there is contention whereas lazy conflict detection remains roughly constant.

## Methodology

This section describes the specific implementation details of the hardware transactional memory systems as well as the simulation environment, methodology, and benchmarks used to evaluate the power and energy characteristics of transactional memory workloads.

### CMP Design

Figure 4-1 shows the basic system architecture and Table 4-6 summarizes the design parameters. The CMP system consists of 4 processing elements based on 65nm technology; the base number of processors was chosen to reflect currently available configurations. The processors are 4-issue out-of-order with a split 64kB 4-way set associative write-back L1 cache. There is a 4MB 8-way set associative shared L2 cache split into 8 banks. The off chip memory is modeled as 4GB of DDR3. Cache coherence is maintained using a snoop-based MESI protocol.

### HTM Design

The HTM simulator is a modified version of SuperTrans [60], a cycle-accurate detailed hardware transactional memory model. Using the MIPS ISA, SuperTrans includes support for eager and lazy versioning and conflict detection modes. The conflict detection and version management schemes in SuperTrans are abstract, meaning while they were guided by previously proposed implementations [24] [39], they do not strictly follow any specific transactional memory design; they are idealized representations. SuperTrans was modified to mimic a generic signature-based

transactional memory system similar to LogTM-SE [85] and BulkSC [10] but with many specific parameters still left to the user. SuperTrans tracks read- and write-sets using per-processor Bloom filters[6], this could be extended to per-thread but there are never more threads than available processing elements and threads are bound to a processor on creation. SuperTrans was further modified so that both versioning schemes implement a cacheable logging structure, which holds the virtual addresses and old (eager) or new (lazy) values of memory blocks modified during a transaction. Wattch [9] was integrated into the simulator to estimate the energy consumption for 64 individual structures per processor plus an additional 18 global structures based on values obtained from CACTI [77].

Table 4-6 lists both the core and transactional model parameters. Conflict detection is carried out per-cache line. The primary/secondary baseline latencies and primary variable latency quantify the latencies associated with a commit or an abort. The primary latency is associated with the long operation for the selected versioning scheme – abort for eager and commit for lazy. The secondary latency is the opposite, it sets the delays for a fast operation – commit for eager and abort for lazy. The baseline latency is the static overhead associated with a transaction (*e.g.* the sum of the bus arbitration and log overhead) and the variable latency is the additional time required for a transaction based on log size.

Signatures are a promising way to remove some of the overhead required by early TM proposals. This work uses the results from Sanchez et al. [67] and Yen et al. [86] for modeling the hardware implementation of signatures. Each 1024b signature is represented as 2 64B SRAMs along with the logic gates necessary to implement the $H_3$

hashing functions. Each hash function consists of $\frac{n}{2}$ 2-input XORs for each bit of the hash and each XOR is assumed to consist of 6 transistors [80]. The dynamic power for each XOR was estimated using the following formula: $\sum_{i=1}^{N} \frac{1}{2} C_i V_{dd}^2 f_i$ where $C_i$ is the output capacitance of the *i*th gate, $V_{dd}$ is the supply voltage, $f_i$ is the switching frequency, and *N* is the total number of gates. The values were estimated using CACTI [77] and the switching frequency was assumed to be the clock frequency, which gives a worst-case estimation.

**Workloads**

For the evaluation, 14 benchmarks from two different benchmarking suites (SPLASH-2 and STAMP) along with 15 synthetic benchmarks were used. While SPLASH-2 provides a good comparison of design points for fine-grained transactions and highly optimized lock-behavior, it is believed that future transactional workloads will also be comprised of coarse granularity transactions that may not be well tuned. To capture this trend, workloads from the STAMP suite (ver. 0.9.6) of transactional benchmarks are used in the evaluation. Since the STAMP suite does not provide lock-based equivalents of the transactional benchmarks, lock versions were generated using the same level of granularity as the transactions. Table 4-7 gives the input set used for each benchmark. All benchmarks were run to completion.

TransPlant [61], a parameterized transactional memory benchmark creation tool, was used to generate the synthetic benchmarks. TransPlant takes a statistical descriptor file as an input and produces C-code that can be compiled and run on a simulator. Table 4-8 describes the first order design parameters that the user can specify. One of the goals of this work is to isolate those program characteristics that

have the largest impact on the power. To accomplish this, the workloads are constructed so that the transactional work, in terms of instruction count and composition, is held constant. While task decomposition in real applications is not straight forward, keeping the total work constant allows variables to be isolated. For example, if work was not held constant, transaction granularity could not be used as an independent variable in these workloads. Unless otherwise noted, transactions are evenly spaced throughout the program, allowing for a direct comparison across dimensions. Each transaction is responsible for at least one unique load and one unique store so that all transactions have at least some chance of conflicting; the probability of a conflict is random for each benchmark. In the granularity experiments, the work is broken down into successively smaller granularities, each representing a point along an axis into which a programmer could decompose the transactional work. Thus, as the granularity of the transactions becomes finer grained, transactions contain fewer instructions but the total number of transactions required to complete the work increases. While TransPlant provides two modes of conflict modeling, a high mode in which the distance between pairs of load/store operations to the same cache line is maximized and a random mode where this distance is randomized, only the random mode is used for the granularity experiments. Finally, it should be noted that since transactional work is calculated on a per-thread basis, trends can be compared across a varying number of processing elements, however the raw total cycle counts will differ based upon the number of threads. As such, all of the results for the synthetic benchmarks are reported as the mean of 50 trials.

## Standard Benchmark Results

This section provides an analysis of the power and performance of the different hardware transactional memory systems using the SPLASH-2 and STAMP benchmarks. Benchmarks are referenced by the abbreviations in Table 4-7. The system designs are referenced as LK – lock, EE – eager conflict/eager versioning, EL – eager conflict/lazy versioning, and LL – lazy conflict/lazy versioning.

### Power Analysis

A cursory examination of the average power, shown in Figure 4-2, reveals minor differences for many of the benchmarks. This is primarily a result of a lack of contention in the SPLASH-2 benchmarks and very large sequential regions for some of the STAMP benchmarks. This behavior is reflected in the cycle breakdown in Figure 4-3. The benchmarks comprised largely of parallel regions are *genome, kmeans,* and *labyrinth* although there are some additional benchmarks such as *bayes* and *ocean* that are worth discussing because of their energy delay product ($EDP = Et = Pt^2$). EDP quantifies the energy-performance tradeoff for each program, shown in Figure 4-4 (EDP shown is normalized to eager-eager case to accentuate differences in the transactional models).

*Bayes*: This is the longest running benchmark and although the power is dominated by sequential regions, *bayes* is comprised of very large critical sections, averaging 87k instructions. This combination makes for very sparse energy concentrations for all of the designs. The average power for all of the transactional models is under 50W and even the lock power is lower than that of many other benchmarks. This is partially because the benchmark has a very long setup time during which only a single processor is active, bringing the entire average down. In the lock-

based version, the execution becomes serialized with multiple processors waiting for a lock release. This means that execution is concentrated in a single processor while the remaining processors spin on the lock variable, only consuming the power required for reading a cache line. The eager conflict detection schemes experience a similar effect; the read and write sets, while large, are small relative to the transaction size with very few actual conflicts. This composition allows the eager conflict detection scheme to resolve most conflicts through NACKs, which only affect bus energy, leaving the remaining structures idle. While the eager schemes can NACK while waiting for a potentially conflicting address, lazy schemes only check for conflicts when a transaction commits. For *bayes* this results in a tradeoff – fewer aborts but the rollbacks are much more costly in terms of execution time. However, this benefits power consumption because these rollbacks are expensive in terms of cycles but only require the L1D, L2, and data buses, resulting in lower power density. The effect of NACKs and aborts are most apparent in the EDP (Figure 4-4). Although the average power of the transactional models is half that of the lock-based scheme, the total energy consumed by the transactional models is nearly 50 times that of the lock-based approach.

G*enome*: As with *bayes*, the lock-based version of *genome* has a higher average power rating than any of the transactional models. *Genome* contains nearly 6k critical sections but less than 1% of them result in aborts for the transactional models, which allows them to make forward progress where the lock-version must wait or to reach a staggered execution state where very few of the transactions actually run in parallel. In this case, the lock-based version has almost twice the number of L1 data reads as the transactional versions and nearly twice the runtime but finishes the same

amount of work, making its EDP nearly 3.5 times higher than the worst performing transactional system. The scenario in *bayes* was that there was a smaller number of critical sections but they were very large. The critical sections in *genome* are two orders of magnitude smaller, so while this does have the effect of lowering the average power, the impact is mitigated by the sheer number of critical regions. Lazy conflict detection has slightly higher average power than eager because it cannot stall to avoid aborts and must rollback and redo more work, just as with *bayes*.

*Kmeans*: This benchmark has the greatest variability in power out of all of the benchmarks but the explanation is subtle. Both eager conflict detection schemes have lower average power than lazy conflict detection because they are able to resolve most conflicts through NACKs, avoiding some of the costly aborts that lazy experiences, in terms of energy, while extending the execution time. Furthermore, the eager-lazy implementation achieves lower average power because it is able to avoid aborts and the time required to copy the values from the log back to the L1 gives it a 15% advantage over eager-eager for the same reasons discussed above. However, from Figure 4-4, this does not translate to better power-performance. The eager-lazy system is able to resolve non-circular conflicts through stalls but if more than one transaction attempts to commit, multiple transactions will be waiting for the data bus to become free. This results in longer stall periods than the eager-eager model but fewer aborts since the transactions reach a steady state in which they are working on disparate transactions that do not conflict. Taken together, this means longer runtime with fewer places where the processors have low utilization leading to a 12% increase in the EDP. The lazy-lazy system suffers from aborts as well as contention for the commit bus,

giving it the highest average power at 190W as well as increasing the EDP to almost 2.5 times that of the eager-eager system.

*Labyrinth*: This benchmark consists of very coarse-grain critical regions, averaging almost 400k instructions each, making it the coarsest of the benchmarks and giving it, from Figure 4-3, the highest ratio of contentious work. The lock-based program is able to make steady forward progress while the transactional implementations suffer from multiple rollbacks and stalls, raising the average power for locks a little higher than the eager conflict detection schemes but giving it a much lower EDP – roughly 15% of that of the eager-eager model. The lazy-lazy scheme suffers from twice as many aborts but after the initial convoying problems is able to abort early. The low number of NACKs and aborts increases the average power because the processor remains active but brings the EDP down to 61% of the eager-eager model.

*Ocean*: Both *ocean-contiguous* and *ocean-noncontiguous* appear to be pedestrian. They are well designed and avoid most contention so they have very small sporadic critical sections (approximately 10 instructions) that account for less than 1% of the total execution. As such, one would expect the models to behave roughly the same. Assessing their behavior based on the average power in Figure 4-2, the transactional models do look equal, but the lock-based program completes these programs almost an order of magnitude faster, reducing their EDP to below 10% of that of the transactional models. Like *bayes*, these two programs magnify the overhead of the transactional models when there is contention. The difference with the ocean benchmarks is that they are comprised of a few very small critical sections. *Bayes'* 87k-instruction transactions with read and write sets approaching 1k brought the average

power down because the overhead for the aborts was so large and the pipeline was stalled for hundreds of cycles at a time. *Ocean*'s average transaction size is about 10 instructions with read- and write-sets fewer than 3 cache lines so aborts recover quickly and stalls are short. This program structure has the effect of increasing the execution time while keeping the per-structure energy roughly constant and increasing the EDP of the transactional models.

*Vacation*: On the surface, this benchmark is similar to *fmm* in that there is no actual contention and the read and write sets are relatively small. However, *vacation*'s transactions are almost 200 times larger than *fmm*'s and, despite its relatively short runtime, the critical sections comprise nearly 15% of its execution time in the transactional models. The lack of contention is reminiscent of *genome*, where the lock program is forced to stall at all synchronization points but the transactional models are able to make forward progress and only wait at barriers; the difference for *vacation* is that, unlike *genome*, the transactions never abort and rarely stall one another. The average power is low for all of the models because, like *bayes*, the majority of execution takes place in the parent thread during initialization.

**Structural Analysis**

An analysis of the energy consumed per structure provides more insight into the impact that the log file and signatures have on the overall power. While the fine-grain power model gives results for 82 architectural structures, the figure above merges many of them for ease of viewing. Except for *genome*, *kmeans*, and *labyrinth*, none of the benchmarks exhibit much deviation (18% on average) in the energy distributions of the lock and transactional models. Based on that fact, a single sample should provide an overview of the generalized behavior. Figure 4-5 shows the structural energy

breakdown for *kmeans*, which was chosen because it has one of the largest differentiations of all of the benchmarks (it is not possible to show the results for all of the benchmarks).

For all of the design points, the load/store queue, register file, and ALUs consume the largest portion of the energy – 76% of the total energy on average. The main difference between the traditional model and the transactional models is the energy consumed by the load and store queues, which require an additional 15-50% additional energy over the lock-based implementation. This is primarily a result of having to rollback and re-execute aborted transactions. Relative to other structures, reading and writing to these queues can be expensive (roughly 7nJ per read/write).

The eager conflict detection schemes have nearly identical energy distributions (this is true for all of the other benchmarks as well). The only difference is in the L2 energy for the lazy versioning scheme. The total energy for the L2 cache increases by an average of 4% for eager-lazy relative to eager-eager because it must read from the log on a commit. The lazy-lazy scheme suffers from the same drawback as eager-lazy but it also aborts more frequently, which puts more pressure on both the L2 and signature hardware.

*Genome* and *vacation* have energy distributions almost identical to that of *kmeans* for all of the models. *Labyrinth* is the only benchmark where the signature energy for the eager conflict detection schemes makes up a larger portion of the total energy than lazy conflict detection, increasing by 14%. This is because the lazy-lazy implementation is able to avoid most conflicts because the threads become discordant, allowing them to execute dissimilar transactions. The remaining benchmarks have energy distributions

that are nearly homogeneous across all of the models because many of the other benchmarks have little to no contention, making the dominate structures the same across all dimensions.

## Synthetic Workload Results

This section provides an analysis of the power and performance of the different hardware transactional memory systems using synthetic benchmarks. Synthetic benchmarks [3] are miniature programs for use in early design evaluations. The advantage of synthetic benchmarks is that they can be used when the simulation time of real benchmarks is prohibitively long or for design space evaluation where no suitable benchmark exists. The methods used in this paper are a parameterized form of workload synthesis [61].

For these experiments, the transactional granularity, the raw size of a transaction, is scaled by powers of 2 beginning with 8 instructions and continuing to 128k instructions; the transaction stride, the distance between transactions, is equal to the transaction size so that the static number of transactional and sequential instructions remains equal. Memory accesses are modeled as circular arrays. On a per-thread basis, there is no reuse outside of the transaction that first references a specific location, ensuring that a single transaction in each thread can only interfere with a single transaction in another thread. For example in a program with *n* threads, $TX_1$-A can interfere with $TX_2$-A, $TX_3$-A, …, and $TX_{n-1}$-A but never with $TX_n$-B, where *n* is the thread ID.

### Power Analysis

Figure 4-6 shows the average power for the synthetic benchmarks as the transaction granularity increases. For each synthetic benchmark, unlike the SPLASH-2

and STAMP benchmarks, there is a distinct variation in the average power between the three models that becomes more pronounced as the granularity increases. Up until the transaction size reaches 4k, the eager-lazy model has the highest average power out of the three designs – peaking at 70W. At that point, there is an abrupt drop in the average power for the two eager conflict detection schemes; eager-eager drops by 54% to 32W and eager-lazy drops by 60% to 41W. The average power for the lazy versioning scheme increases by an average of 6% until the granularity reaches 32k at which point it begins slowly decreasing. To explain these phenomena the breakdown of both the transactional cycles, shown in Figure 4-8, must be analyzed.

The top graph in Figure 4-8 shows the relative execution time for the eager-eager model. From Figure 4-6, the average power remains flat until 128-instruction transactions. This is because the processor spends a majority of its time performing useful sequential or transactional work. The average power does not increase as the overhead decreases because, although the bookkeeping overhead (bus arbitration etc.) can result in pipeline stalls that reduce the average power, the stalls are short lived. At 128-instructions, there is a slight uptick because the transactional work outweighs the overhead. The trend continues until the transaction size reaches 4k, at which point the system begins to experience contention. The eager-eager system is able to avoid aborting by stalling the processor, which reduces the average power because the processor becomes idle. The increased power at consumption at 32k is because the system can no longer completely avoid aborting so some transactions must be reissued. This lower average power has a price though in the performance domain. Figure 4-7 shows how the EDP changes as the transaction granularity increases for all

three transactional systems. As can be seen, the EDP for the systems is both equal and flat, averaging 3e10$Ws^2$, until the 4k mark when the eager versioning systems begin taking a performance hit, increasing the EDP more than 35 times, due to contention.

The eager conflict detection/lazy versioning behavior is shown in the middle graph of Figure 4-8. Until the transaction size reaches 4k, the average power remains roughly constant because the processors are able to spend almost 100% of the time performing useful work; there are no aborts and few stalls. The 512 and 1k benchmarks accentuate the added power overhead for lazy versioning when the average power of the eager conflict detection schemes are compared. At this point, both eager conflict detection schemes spend almost 100% of their time performing useful work but the eager-lazy model must update the log during transactions and copy the log back on a successful commit, whereas the eager-eager model only needs to clear the log pointer at commit. At 4k, aborts and stalls completely overtake successful execution causing a 40% drop in the average power, from 67W to 41W. This figure also shows that the extra time spent executing aborted work pushes its power consumption higher than that of eager-eager but even a moderate amount of time spent in the stall state (10-20%) can have a large impact on the average power for a processor.

The lazy-lazy cycle breakdown is shown in the bottom graph of Figure 4-8. There is more contention for lazy conflict detection than eager. For the 8- and 16-instruction transactions, the average power for is lower than the other two by more than 30%. This is because the execution becomes serialized as transactions are waiting to commit, resulting in idle time for the processors. The serialization potential quickly diminishes as the transactions begin overlapping and aborts begin occurring. Overall, the average

power for this system increases steadily, between 2% and 20% at each step as the processors remain constantly busy and there are fewer opportunities for stall events. Moreover, while the average power increases, the EDP for lazy-lazy remains almost constant, indicating that lazy-lazy represents a worst-case implementation.

## Related Work

Herlihy and Moss ) [26] began the transactional memory resurgence and since then, the architecture community has been racing to provide new implementations and tweak existing ones. Transactional Coherence and Consistency (TCC) [24] was one of the first models to use transactional memory and works under the assumption that transactions constitute the basic block for all parallel work, communication, coherence, and consistency. TCC uses a lazy-lazy approach, which makes aborts fast but commit time relatively slow. LogTM was different in that the designers chose to make commits fast and aborts slow by storing stale data values in a per-thread log. The assumption being that commits will be more frequent than aborts in typical applications. Architects seem to have become fixated on these specific designs without much considering for the power implications. Ceze *et al*. [11] first proposed signatures for use in transactional memory and they were quickly adopted by LogTM-SE [85] and SigTM [67]. In these systems, during a transaction, load and store operations insert addresses into read and write signatures. The signatures are cleared on a successful commit or abort operation. In [7], Bobba *et al.* explored different performance pathologies that can arise across different transactional models. This work has benefited from that study by incorporating several of the fixes to the protocols in an effort to provide a fair cross-dimension comparison.

There has been some recent research into the energy use of transactional memory for embedded and multiprocessor systems. Ferri recently proposed unifying the L1 and transactional cache in an embedded system and showed that using a small victim cache to reduce the pressure from sharing improved the energy-delay product [21]. Moreshet *et al.* [53] showed that hardware transactional memory systems can be more energy efficient than locks in the absence of contention. They then proposed a serialization mechanism for HTMs and showed that it lowered energy consumption for their microbenchmarks. However, their work relied on four non-contentious SPLASH benchmarks and one in-house microbenchmark, making it difficult to draw any meaningful conclusions. Using an analytical model to estimate the additional power for an Alpha 21264, Sanyal *et al.* [69] proposed a technique for clock gating on an abort using TCC. This work differs in that it is focused on characterizing the power implications of transactional memory using a comprehensive approach.

## Summary

Although there have been many proposed transactional memory designs, few have focused on the power-performances aspects of transactional memory and there has been no systematic evaluation of how transactional design decisions affect system power. Marginalizing the energy consumption of the proposed architectures could adversely affect design decisions and force future research along the wrong path. This research uses the SPLASH-2 and STAMP benchmark suites as well as synthetic workloads to analyze the power and energy for three different transactional systems: eager conflict/eager versioning, eager conflict/lazy versioning, and lazy conflict/lazy versioning.

There is no measurable difference in the average power consumption of the SPLASH-2 benchmarks for the lock and eager conflict detection schemes but the average power for lazy conflict detection is slightly higher due to increased log utilization. The energy delay product (EDP), used to merge the power and performance domains, for the SPLASH-2 benchmarks is reflective of the average power for all but the *ocean* benchmarks, which magnify the increased power consumption and execution overhead of having the transactional models implemented in hardware – increasing their EDP up to 21 times that of the lock-based implementation. For the STAMP benchmarks, the average power varies from 32W to 192W. While any model would suffice for the SPLASH-2 benchmarks, eager-lazy minimizes the average power for the STAMP benchmarks. However, the EDP for these benchmarks suggests that because their behavior is so diverse, there is no clear design choice that minimizes the power-performance. For benchmarks with little actual contention but many critical sections, such as *genome*, locks have higher EDP than any of the transactional models. If benchmarks have real contention, like *kmeans* and *labyrinth*, locks have lower EDP than any of the transactional models. By using synthetic benchmarks, scaling the transaction size and, indirectly, the contention, it is shown that the average power of the three transactional systems does not vary substantially. With heavy contention, eager conflict detection can reduce the average power by 50%. When the benchmark performance is taken under consideration in conjunction with power, eager conflict detection schemes can be unpredictable whereas the EDP of lazy conflict detection schemes remains nearly constant, never varying more than 4%.

Experimental results show that when there is little or no contention, hardware transactional memory consumes approximately the same amount of power as a lock-based system. Under moderate or heavy contention, some of the transactional memory designs have much lower average power than their lock counterparts but this does not always correspond to a better design choice given that the energy-delay product is often worse than that of the lock version. The conclusions drawn from the SPLASH-2 and STAMP benchmarks are validated using a systematic evaluation of synthetic benchmarks.

Table 4-6. Baseline Configuration

| | | Parameters |
|---|---|---|
| Core Model | Processing Elements | 2.4GHz, out-of-order, 4-issue, 65nm |
| | L1D Cache | 32kB, 4-way, 32B blocks, 2-cycle latency |
| | L2 Cache | 4MB, 8-way, 32B blocks, 14-cycle latency |
| | Off-chip memory | 240 cycle latency |
| Transactional Model | Conflict Detection | Eager and lazy |
| | Version Management | Eager and lazy |
| | Conflict Resolution | Requester/Committer wins with exponential backoff |
| | Conflict Granularity | 32B |
| | Primary Baseline | 50 |
| | Primary Variable | 12 |
| | Secondary Baseline | 12 |

Table 4-7. Benchmark Parameters

| Benchmark | Abbreviation | Input |
|---|---|---|
| *barnes* | BN | 16K particles |
| *bayes* | BY | 1024 records |
| *cholesky* | CH | tk15.O |
| *fluidanimate* | FA | 35kMips |
| *fmm* | FM | 16K particles |
| *genome* | GN | g256 s16 n16384 |
| *kmeans* | KM | Random1000_12 |
| *labyrinth* | LB | x32-y32-z3-n96 |
| *ocean-con* | OC | 258x258 |
| *ocean-non* | ON | 66x66 |
| *raytrace* | RT | Teapot |
| *vacation* | VA | 4096 tasks |
| *water-nsq* | WN | 512 molecules |
| *water-sp* | WS | 512 molecules |

Table 4-8. Transactional- and Microarchitecture-Independent Characteristics

| Characteristic | Description |
|---|---|
| Threads | Total number of threads in the program |
| Homogeneity | All threads have the same characteristics |
| Tx Granularity | Number of instructions in a transaction |
| Tx Stride | Number of instructions between transactions |
| Read Set | Number of unique reads in a transaction |
| Write Set | Number of unique writes in a transaction |
| Shared Memory | Number of global memory accesses |
| Conflict Dist. | Distribution of global memory accesses |
| Tx Inst. Mix | Instruction mix of transactional section(s) |
| Sq Inst. Mix | Instruction mix of sequential section(s) |

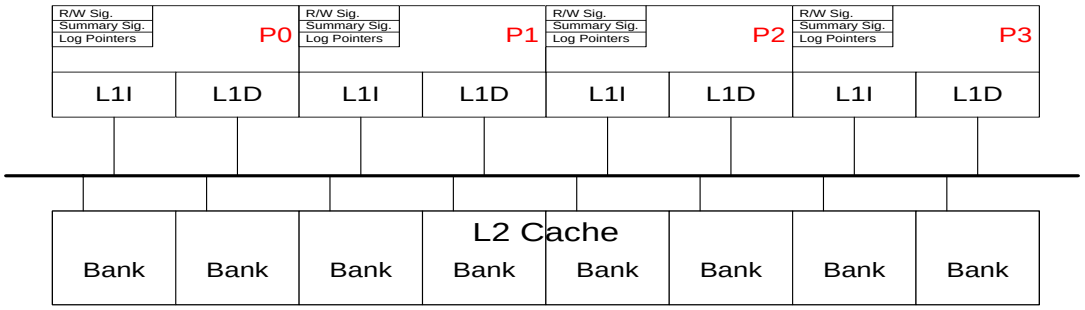


Figure 4-12. Baseline CMP Design

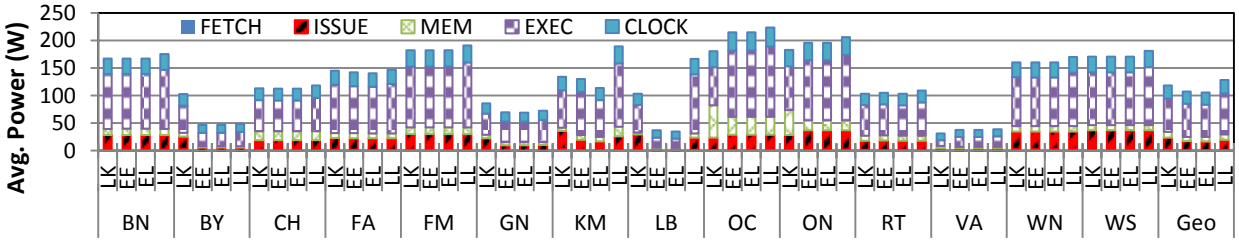


Figure 4-13. Real Benchmark Power

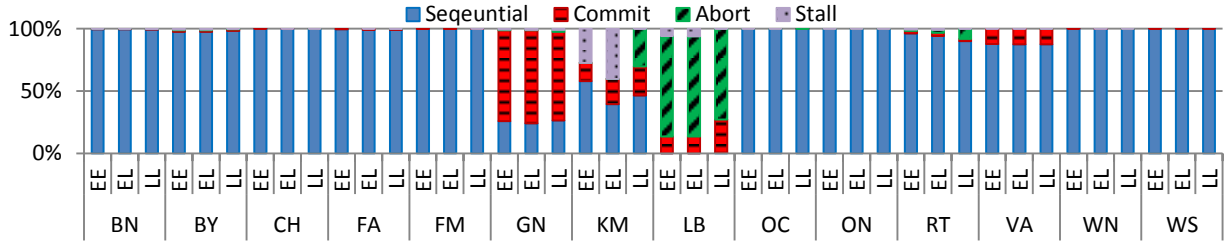


Figure 4-14. Cycle Breakdown by Execution Type For Real Benchmarks

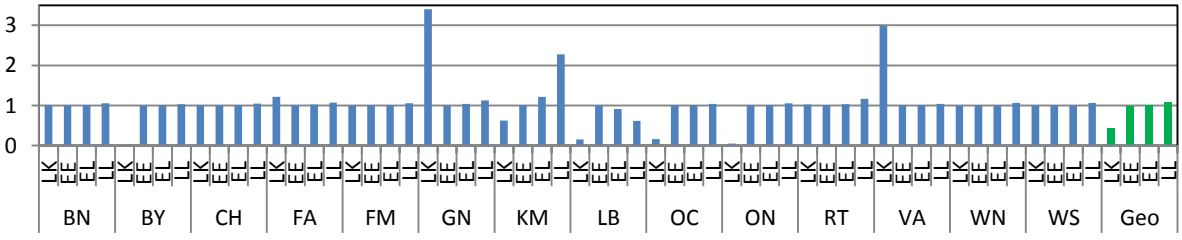


Figure 4-15. EDP ($Pt^2$) Normalized to Eager Versioning/Eager Conflict Detection (EE)

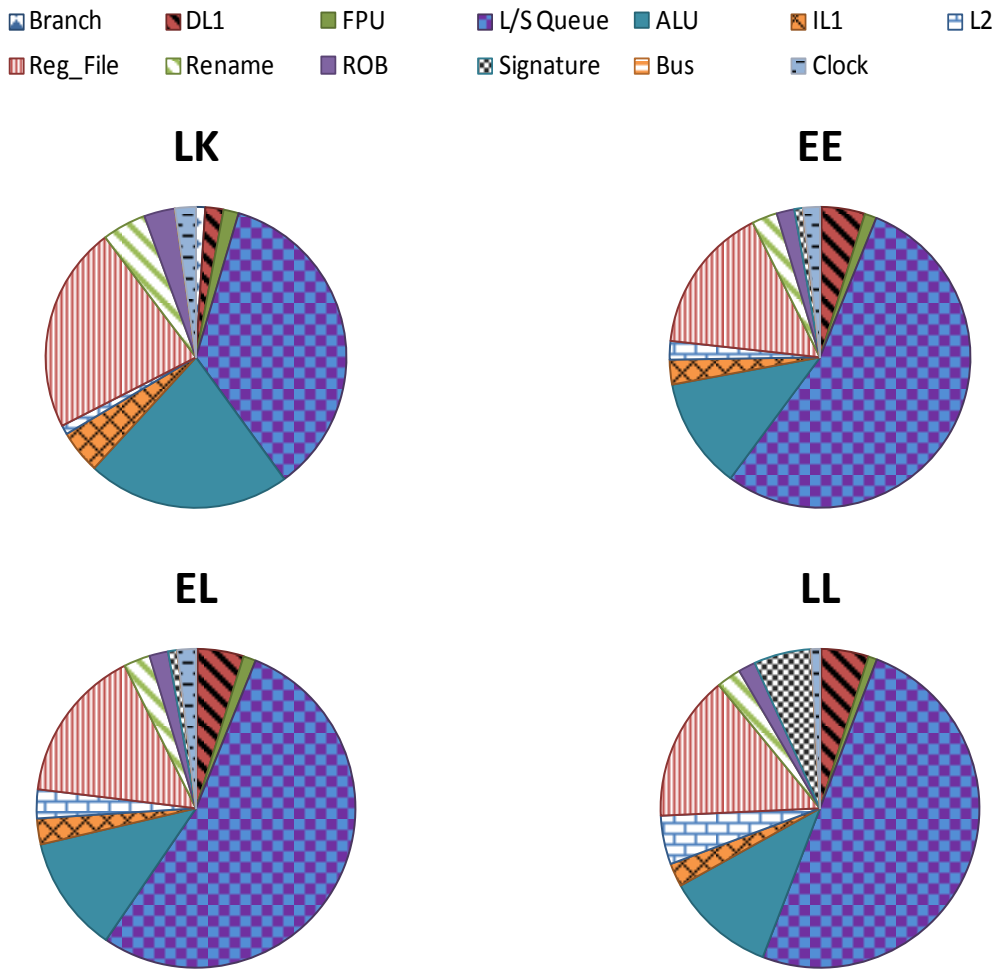


Figure 4-16. Average Per-Structure Energy

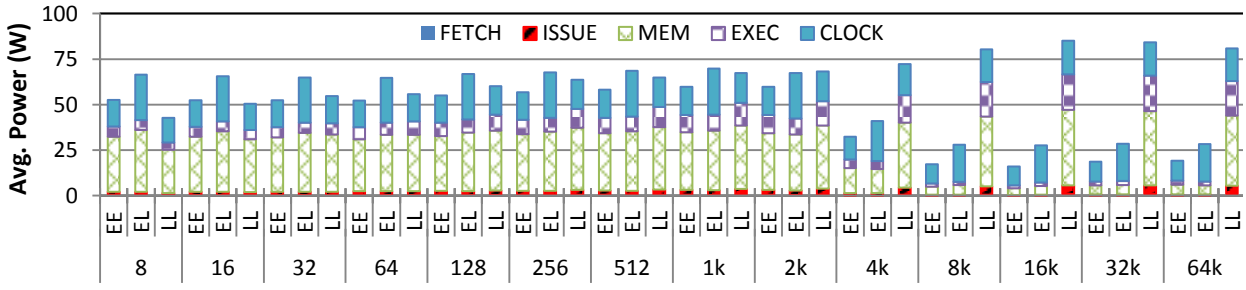


Figure 4-17. Synthetic Benchmark Power

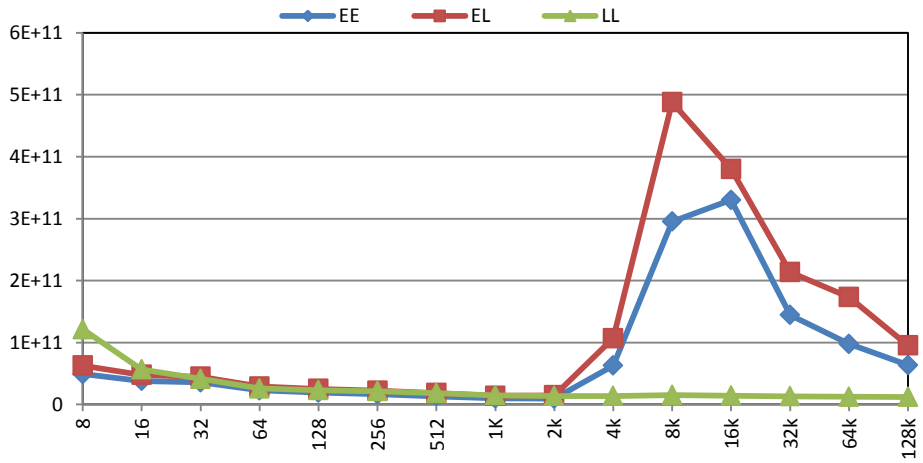


Figure 4-18. Synthetic EDP ($Pt^2$)

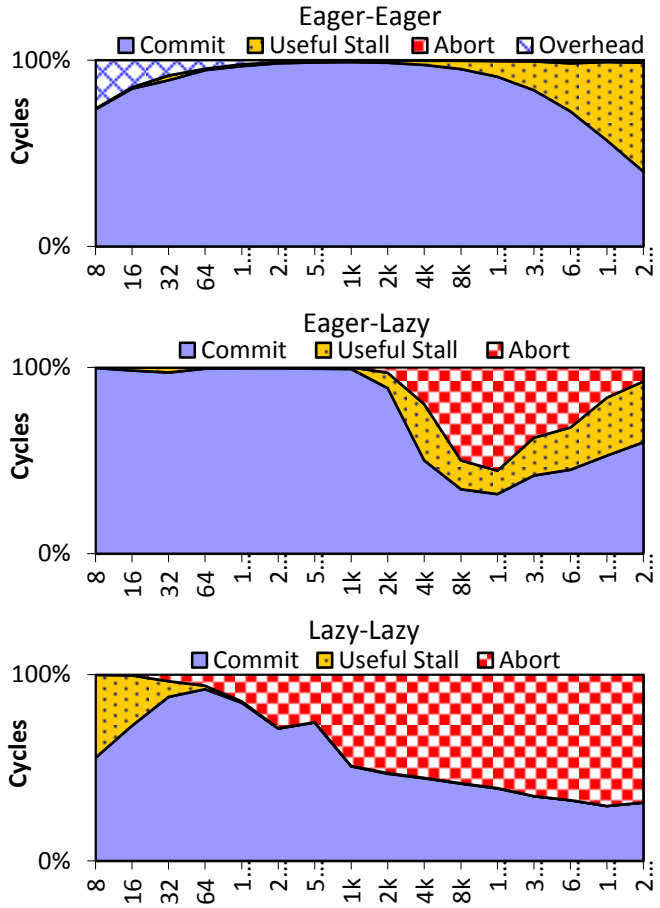


Figure 4-19. Relative Execution Time

# CHAPTER 5
# OPTIMIZING THROUGHPUT/POWER TRADEOFFS IN HARDWARE TRANSACTIONAL MEMORY USING DVFS AND INTELLIGENT SCHEDULING

## Background and Motivation

Power dissipation continues to be a first-order design constraint for modern computer designs from the chip level to data centers. At the chip level, power consumption can affect its reliability and performance and can increase packaging and manufacturing costs. And while chip multi-processors (CMPs) offer better energy efficiency than previous uniprocessors [42], they still suffer from the same heat removal problems as previous generations. However, unlike previous generations, CMPs provide more opportunities for balancing energy use. CMPs are designed for running multiple threads of execution, which often vary in performance and resource requirements, making them ideal candidates for runtime optimizations that can maximize program performance while minimizing chip power consumption.

However, most of the threads are actually disparate processes each running with a single thread. In order to exploit the types of resources offered by CMPs, programmers will need to begin changing the way they write code, writing programs that consist of multiple threads that are able to take advantage of the ever-increasing number of processing elements (PEs). However, exploiting the available data and task parallelism in a program is often a challenging and time-consuming process, requiring significant time investments to extract performance and guarantee correctness. Transactional memory (TM) [26] has been proposed as a programming technique to replace locks, shifting some of the burden of synchronization from the programmer to the system architecture – in effect providing an abstraction of the implementation. While transactional memory was conceived of as a means to shift the programming burden,

hardware implementations have ancillary benefits such as increased performance over locks and potential energy savings [53].

This work focuses on how transactional memory can be leveraged for energy and performance optimizations, making the following contributions:

1. Dynamic frequency and scaling (DVFS) is introduced to reduce the power consumption of stalled processing elements and increase overall throughput by setting the clock frequency and supply voltage for each PE based on its current execution state and those of the collocated PEs. The optimization decreases the amount of **time** that a processor holds its read and write sets by increasing the clock frequency of NACKing PEs and decreasing the **power consumption** of NACK'd PEs by throttling the clock frequency. Using this DVFS policy improves the energy delay product (EDP), which is a joint measurement of the system power and performance, by up to 43%.
2. A new preemptive transaction scheduler is introduced based on the system's conflict density. This scheduling policy prevents potentially contentious transactions from issuing and clock gates the resident PE, reducing the system power. Furthermore, because there are fewer executing transactions those that remain running have a lower probability of experiencing an abort, which increases total throughput and reduces the system EDP by as much as 76%.
3. The new policies are then combined and compared against previously proposed power management techniques for transactional memory that use clock gating and transactional serialization. The new policies show improvements between 12% and 30% relative to the previous work.

## Motivation

This section provides an overview of the power of the different hardware transactional memory systems using the SPLASH-2 [83] and STAMP [50] benchmarks. Benchmarks are referenced by the abbreviations in Table 5-3 and the transactional memory systems are described using their primary design points: conflict detection and version management. Conflict detection defines when conflicts are detected and version management defines where new and old values within a transaction are stored. Both use the same basic nomenclature and can be either eager or lazy. With eager conflict detection, addresses are checked for conflicts on each read and write within the transaction whereas lazy checks addresses when a transaction attempts to commit. Eager version management writes new values in place and copies old values elsewhere; lazy does the opposite, leaving old values in place and writing new values elsewhere. The system designs are referenced as LK – lock, EE – eager conflict/eager versioning, EL – eager conflict/lazy versioning, and LL – lazy conflict/lazy versioning.

Consider Figure 5-1, which shows a breakdown of the power consumption from 14 benchmarks using locks, eager-eager, eager-lazy, and lazy-lazy. There is no measurable difference in the average power consumption of the SPLASH-2 benchmarks for the lock and eager conflict detection schemes but the average power for lazy conflict detection is slightly higher due to increased log utilization. For the STAMP benchmarks, the average power varies from 32W to 192W. While any model would suffice for the SPLASH-2 benchmarks, eager-lazy minimizes the average power for the STAMP benchmarks. However, the EDP for these benchmarks suggests that because their behavior is so diverse, there is no clear design choice that minimizes the power-performance. These experimental results show that when there is little or no

contention, hardware transactional memory consumes approximately the same amount of power as a lock-based system. Under moderate or heavy contention, some of the transactional memory designs have much lower average power than their lock counterparts. What is important is that the slack power available in these benchmarks can be exploited to improve the performance while limiting the maximum chip power and temperature.

## Methodology

This section describes the specific implementation details of the hardware transactional memory systems as well as the simulation environment, methodology, and benchmarks used to evaluate the power and energy characteristics of transactional memory workloads.

### CMP Design

Figure 5-2 shows the basic system architecture and Table 5-1 summarizes the design parameters. The CMP system consists of 4 processing elements based on 65nm technology; the base number of processors was chosen to reflect currently available configurations. The processors are 4-issue out-of-order with a split 64kB 4-way set associative write-back L1 cache. There is a 4MB 8-way set associative shared L2 cache split into 8 banks. The off chip memory is modeled as 4GB of DDR3. Cache coherence is maintained using a snoop-based MESI protocol. The power management structures are discussed in Section 5-3.2.

### Simulator Design

The transactional memory simulator is a modified version of SuperTrans [60], a cycle-accurate detailed hardware transactional memory model that includes support for eager and lazy versioning and conflict detection modes. The conflict detection and

version management schemes in SuperTrans are abstract, meaning while they were guided by previously proposed implementations [24] [52], they do not strictly follow any specific transactional memory design; they are idealized representations. SuperTrans was modified to mimic a generic signature-based transactional memory system similar to LogTM-SE [85] and BulkSC [10] and tracks read- and write-sets using per-processor Bloom filters [6]. Both versioning schemes implement a cacheable logging structure, which holds the virtual addresses and old (eager) or new (lazy) values of memory blocks modified during a transaction.

Table 5-1 lists both the core and transactional model parameters. Conflict detection is carried out per-cache line. The primary/secondary baseline latencies and primary variable latency quantify the latencies associated with a commit or an abort. The primary latency is associated with the long operation for the selected versioning scheme – abort for eager and commit for lazy. The secondary latency is the opposite; it sets the delays for a fast operation – commit for eager and abort for lazy. The baseline latency is the static overhead associated with a transaction (*e.g.* the sum of the bus arbitration and log overhead) and the variable latency is the additional time required for a transaction based on log size.

The signature implementation uses the results from Sanchez *et al.* [67] and Yen *et al.* [86] for modeling the hardware implementation of signatures. Each 1024b signature is represented as 2 64B SRAMs along with the logic gates necessary to implement the $H_3$ hashing functions. Each hash function consists of $\frac{n}{2}$ 2-input XORs for each bit of the hash and each XOR is assumed to consist of 6 transistors [81]. The dynamic power for each XOR was estimated using the following formula: $\sum_{i=1}^{N} \frac{1}{2} C_i V_{dd}^2 f_i$

where $C_i$ is the output capacitance of the *i*th gate, $V_{dd}$ is the supply voltage, $f_i$ is the switching frequency, and *N* is the total number of gates. The values were estimated using CACTI [77] and the switching frequency was assumed to be the clock frequency, which gives a worst-case estimation.

The power management system was modeled after Intel's Foxton Technology (FT) [49] and includes on-chip power and temperature sensors and a small microcontroller. Internally, the microcontroller was modeled as a single structure that consumes 0.5% of the total chip power. DVFS was added to SESC with the levels shown in Table 5-2. Wattch [9] was integrated into the simulator to estimate the energy consumption for 64 individual structures per processor plus an additional 18 global structures based on values obtained from CACTI [77]. HotSpot [75] was used to estimate on-chip temperature, which is based on the current chip power and feeds into HotLeakage [89] to estimate the leakage power. Although recent work has explored the feasibility of on-chip regulators [40], this work assumes that voltage transitions require approximately 50k cycles at the base frequency or 200ns. When down-scaling the DVFS level, the frequency drop occurs over a two cycle period with the voltage lagging behind over the transition period. Up-scaling the DVFS level increases the frequency and voltage simultaneously over the transition period.

**Workloads**

For the evaluation, 14 benchmarks from two different benchmarking suites (SPLASH-2 and STAMP) along with 15 synthetic benchmarks were used. While SPLASH-2 provides a good comparison of design points for fine-grained transactions and highly optimized lock-behavior, it is believed that future transactional workloads will also be comprised of coarse granularity transactions that may not be well tuned. To

capture this trend, workloads from the STAMP suite (ver. 0.9.6) of transactional benchmarks are used in the evaluation. Since the STAMP suite does not provide lock-based equivalents of the transactional benchmarks, lock versions were generated using the same level of granularity as the transactions. Table 5-3 gives the input set used for each benchmark. All benchmarks were run to completion.

TransPlant [61], a parameterized transactional memory benchmark creation tool, was used to generate the synthetic benchmarks. TransPlant takes a statistical descriptor file as an input and produces C-code that can be compiled and run on a simulator. Table 5-4 describes the first order design parameters that the user can specify. One of the goals of this work is to isolate those program characteristics that have the largest impact on the power. To accomplish this, the workloads are constructed so that the transactional work, in terms of instruction count and composition, is held constant. While task decomposition in real applications is not straight forward, keeping the total work constant allows variables to be isolated. For example, if work was not held constant, transaction granularity could not be used as an independent variable in these workloads. Unless otherwise noted, transactions are evenly spaced throughout the program, allowing for a direct comparison across dimensions. Each transaction is responsible for at least one unique load and one unique store so that all transactions have at least some chance of conflicting; the probability of a conflict is random for each benchmark. In the granularity experiments, the work is broken down into successively smaller granularities so that as the granularity of the transactions becomes finer, transactions contain fewer instructions but the total number of transactions required to complete the work increases proportionately. While

TransPlant provides two modes of conflict modeling, a high mode in which the distance between pairs of load/store operations to the same cache line is maximized and a random mode where this distance is randomized, only the random mode is used for the granularity experiments. Finally, it should be noted that since transactional work is calculated on a per-thread basis, trends can be compared across a varying number of processing elements, however the raw total cycle counts will differ based upon the number of threads. As such, all of the results for the synthetic benchmarks are reported as the mean of 50 trials.

## Using Scheduling and DVFS for Improved Power-Performance

The discussion in Section 5-2 suggests that aborts and stalls have a large impact on the power and performance of many of the benchmarks. If true then there should be a net power-performance gain by avoiding time-intensive aborts and stalls. The first proposed policy leverages dynamic voltage and frequency scaling (DVFS) to decrease the amount of time processing elements are stalled during a NACK. A second policy, based on transaction scheduling, is proposed that utilizes a transaction's current conflict density [30] and its past performance to determine whether a transaction should be preemptively stalled, reducing a program's contention. A further extension, clock gating, is used to reduce the dynamic power of the stalled transaction.

### Using DVFS to Improve Transaction Throughput

Dynamic voltage and frequency scaling (DVFS) was introduced [46] as a means to reduce system power by dynamically controlling the voltage and frequency of PEs based on the system load. DVFS can be implemented at many levels within a system – in the microarchitecture [48], the operating system [33], or at the compiler level [28]. In

this work, the power controller is modeled as Intel's FT controller and embedded in the microarchitecture.

On each 2µs probe interval, the conflict manager is queried. If a stall is detected, the DVFS manager is invoked and the stalled core's frequency is decreased by 266MHz while the stalling core frequency is increased by 133MHz until the upper and lower bounds are reached, at 2.93GHz and 1.2GHz, respectively. If there are multiple stalled transactions residing on multiple processing elements, then the processor frequency is increased an additional step for each stalled processing element. On a successful commit, the power manager is preempted and all processing elements are returned to their default operating frequency. In the event of an abort, the process is repeated. However, if the abort count exceeds some allowable threshold, the aborted processing element is put into an idle state. While in this state, the core's clocks are gated (phase locked loops are disabled) and its caches flushed. The aborting processing element is then assigned to the highest performance state. The processing element remains at this frequency unless the chip-wide power approaches its threshold or unless there is a thermal emergency. On a successful commit, it returns to its default operating frequency and sends a signal to wake the idle processing element. By relaxing the contention between the transactions and exploiting the newly available slack power, total throughput is increased while maintaining or reducing average chip-wide and per-processing element power beneath the package’s allowed electrical and thermal limits.

**DVFS Results**

Figure 5-3 shows the EDP ($Et^2$) of the transactional execution normalized to the baseline of each implementation when using the dynamic voltage and frequency scaling scheme described in Section 5-4.1. As can be seen from the figure, the proposed

scheme improves the EDP by 8% for eager-eager, 7% for eager-lazy, and 7% for lazy-lazy. Because only one processor is allowed to have an up-scaled frequency and multiple processors can be down-scaled, much of the improvement comes from a reduction in energy consumption. There is a greater improvement in the EDP for benchmarks that spend long periods of time with multiple processors in a NACK'd state such as *bayes*, *kmeans*, and *labyrinth*, but the reasons for the improvements can be applied to the remaining benchmarks.

*Bayes*: This is the longest running benchmark and is comprised of very large critical sections, averaging 87k instructions. However, the parent thread has a long setup time and skews cycle calculations, making it appear that there is very little contention. Once the program reaches the parallel regions, *bayes* is highly contentious. The read and write sets, while large, are small relative to the transaction size with very few circular conflicts. This composition allows the eager conflict detection scheme to resolve most conflicts through NACKs, which benefits from the DVFS policy. While the eager schemes can NACK while waiting for a potentially conflicting address, lazy schemes only check for conflicts when a transaction commits. The DVFS policy considers contention for the commit bus as a NACK, which along with the abort policy is why there is moderate improvement for the lazy conflict detection scheme but less so, for all benchmarks, than the eager conflict detection schemes.

*Kmeans*: This benchmark has the highest ratio of stall cycles to total cycles of all of the benchmarks. The DVFS policy reduces the average power of eager-eager by 13%, eager-lazy by 14%, and lazy-lazy by 16% while reducing the execution time by 2% in all cases. For *kmeans*, lazy-lazy obtains more benefit because, for the baseline case,

the transactions suffer from contention for the commit bus as well as aborts, extending the execution time and increasing the energy consumption due to rollbacks. The DVFS policy helps by reducing the number of aborts from 412 to 278, decreasing the total energy and the execution time.

*Labyrinth*: This benchmark consists of very coarse-grain critical regions, averaging almost 400k instructions each, making it the coarsest of the benchmarks and giving it the highest ratio of contentious work – both in terms of aborts and stalls. All of the transactional implementations suffer from multiple rollbacks and stalls. This is the only benchmark where the average power increases with the proposed DVFS. From Figure 5-1, the average power for the eager conflict detection schemes is 30W. The power is low because rollbacks are expensive in terms of cycles but only require the L1D, L2, and data buses, resulting in lower power density. By scaling the frequency and allowing one thread to complete faster than others many of the aborts are avoided (36%), which has a twofold effect. First, the power density is higher because the pipeline is active more often. Second, because there is not as much time spent performing bookkeeping and rollbacks, the execution time is shortened, which increases the power density but decreases total execution time. However, despite the fact that the average power increases to 62.2W for both eager schemes, it remains low enough that there is never a thermal emergency. The average power for the lazy-lazy platform decreases by 26%. The reduction is primarily due to the decrease in aborts with a small decrease in the average power consumption.

While these three benchmarks show the most improvement, the causes of the reduction in EDP can be extended to all of the benchmarks, to some degree. The

average power of *cholesky*, *fmm*, *ocean-contiguous*, *vacation*, and *water-spatial* remains roughly the same for all of the transactional models when using DVFS but the execution time is reduced. For the remaining benchmarks, the average power is reduced along with the execution time (with the exception of *labyrinth* when using eager conflict detection).

**Conflict Probability**

While the DVFS policy discussed in Section 5-4.1 primarily targeted NACKing transactions, the preemptive stalling policy is targeted at aborting transactions and perceived contention within the transactional system. When a transaction aborts, the contention manager resolves the conflict using the prescribed resolution policy. In the systems discussed in Section 5-2, the contention manager invokes an exponential backoff policy that prevents a transaction from reissuing using an exponentially increasing interval, up to some maximum. The proposed addition to the contention manager is called when a transaction begins its execution and works in conjunction with the contention manager. A software manager is invoked within the power controller to compute the transaction's conflict potential for the current iteration, $C[n]$, given by $C[n] = \alpha C[n-1] + (1-\alpha)(C_p)$. Where the conflict probability, $C_p$, is $C_p = (\frac{CurrentAborts}{CurrentAborts + \beta} * \frac{ActiveTransactions}{TotalAvailableProcessors})$. $\alpha$ and $\beta$ are scaling factors used to weight the effect of the previous conflict potential and to determine how responsive the system is to the number of aborts, respectively. If the conflict potential exceeds some threshold, $\rho$, then the transaction is preempted and stalled for a brief interval before it attempts to reissue. If the potential is below the threshold, the transaction is allowed to issue normally. When a transaction begins, a software manager is invoked on the on-

chip microcontroller to calculate the new conflict probability. In the simulator, this is modeled as seven floating-point instructions that must be completed before the transaction begins. The result is stored in a special register in the calling PE. Clock gating is instant while wake-up from clock gating takes two cycles.

Initial tests showed a minor improvement in the EDP for lazy conflict detection but almost no change for eager. This was because eager is already adept at avoiding many of the aborts that affect the lazy implementation and, although there was measurably reduced energy for some of the more contentious benchmarks, much of the improvement in lazy came from reduced runtime. To improve the results, clock gating was introduced to work in tandem with the contention manager and the new scheduling policy. The new scheme works the same as above with two modifications. First, when a transaction is stalled, the processor's clocks are halted, effectively setting the dynamic power to zero for the processor on which the transaction is executing. Second, the processor does not wakeup after a given interval, instead it waits for another transaction to commit before un-gating occurs.

**Conflict Probability Results**

This evaluation is based on the same configurations from Section 5-2 with the addition of the conflict probability scheme. Figure 5-4 shows the new EDP using the scheduling enhancement normalized to the base case for each design point. Improvement in the EDP is seen for all but two benchmarks, *genome* and *raytrace* on the lazy-lazy platform. For most of the benchmarks, the reduction in EDP is the same across all of the transactional implementations, which is due to the lack of contention in the benchmarks. However, the scheduling scheme does reduce both the static and dynamic power of the benchmarks. On average, the proposed scheduling policy

produces in a 6% decrease in the static power due to the reduction in execution time and a 9% decrease in dynamic power because of the clock gating scheme. The reduction in energy use and execution times leads to an EDP reduction of 17% for eager-eager, 17% for eager-lazy, and 10% for lazy-lazy.

Of the benchmarks, *labyrinth* shows the largest improvement across all three implementations while the remainder of the applications show modest improvements. The reason is that *labyrinth* spends more than 98% of its cycles in a NACK or abort state while the other benchmarks typically spend less than 1% of their time in these states. This benchmark consists of very coarse-grain critical regions, averaging almost 400k instructions each, making it the coarsest of the benchmarks and giving it the highest ratio of contentious work. The transactional implementations suffer from multiple rollbacks and stalls and the lazy-lazy scheme suffers from twice as many aborts. All three schemes obtain more than a 2x EDP improvement for *labyrinth* with the eager conflict detection schemes reaching a 5x improvement. For lazy-lazy, the total runtime remains roughly the same but there are 78% fewer aborts, which means that the dynamic power used for speculative execution of these transactions has been saved through preemption and clock gating. For the eager conflict detection schemes, preemptive stalling provides more than 50% reduction in runtime, which directly reduces EDP.

The outliers on the lazy conflict detection scheme, *genome* and *raytrace*, are due to the restrictive scheduling algorithm. Although their total energy is lower than in the base cases, the execution time for these benchmarks is increased by several million cycles, which leads to the increased EDP. For example, *genome* contains nearly 6k

critical sections but less than 1% of them result in aborts for the transactional models. The critical sections in *genome* average 2.4k instructions over 4.9k cycles and comprise 70% of the dynamic execution. When the algorithm is applied to *genome*, the number of aborts is reduced from 106 to 87 and the number of NACKs is reduced from 3794 to 2988 but the average number of cycles consumed by each transaction increases to 5.7k. The algorithm does not consider individual transactions, meaning that it only knows that there are *t* active transactions and not the program counter of each transaction. If each available processor has an active transaction and if the abort count increases too quickly, the result is an overly pessimistic representation of the contention, stalling transactions longer than may be necessary. The end result of which is akin to the serialization scheme discussed in Section 5-4.6. The scaling factors, $\alpha$ and $\beta$, are fixed; a feedback mechanism that can shift these for each active process may provide a better prediction mechanism but the philosophy behind both of the proposed designs was to provide a very simple implementation with very little runtime overhead.

**Combining The Schemes**

Although both of the policies described in Sections 5-4.1 and 5-4.3 are linked with aborts, the DVFS policy relies on NACKs as the primary motivator while the preemptive scheduler relies on perceived contention, allowing the schemes to be use together. Figure 5-5 shows the EDP when both DVFS and the probabilistic conflict scheduler are used together. The proposed policies effectively work together, providing a reduction in the EDP of 19% for eager-eager, 20% for eager-lazy, and 15% for lazy-lazy. The trend is similar to that of Figure 5-4 because the contention management policy provides the majority of the energy reduction for most of the benchmarks. The

exceptions are *barnes*, *bayes*, and *raytrace* (and *genome* for lazy-lazy), which benefit more from the DVFS policy.

**Measuring Up**

In this section, the proposed DVFS and scheduling policies are compared with two previous studies. The first comparison is based on work done by Sanyal *et al.* [68]. When a transaction is aborted by a committing transaction, the clocks of the aborted processor are halted and remain so until a local timer expires. The timer value is derived from an equation that takes into account the abort count and how long the blocked processor has been gated. For the experiments presented here, the model is ideal – meaning that the structures proposed in their work are not modeled at the microarchitecture level and the delay algorithm is able to complete instantly. It should be noted that the original paper used an analytical model to derive results based on memory traces, not integrated functional and timing models. The second comparison is from Moreshet *et al.* [53] who proposed a serialization algorithm for power-savings in hardware transactional memory. When a conflict is detected and a transaction is forced to abort, instead of reissuing the transaction it is placed in a queue until a successful commit is detected at which point it is reissued. When the queue is empty, the system returns to its default state. For their work, the authors only reported the power of the memory subsystem; the results reported here are for the entire processor and main memory.

Table 5-5 provides the results for both the gating (**gating**) and serialization (**serial**) schemes along with the proposed DVFS and scheduling (**DVFS+CS**) policies proposed in this paper. Clock gating alone does not noticeably reduce the EDP for most of the benchmarks. Although the average power of the benchmarks is reduced by an

average of 0.9% for eager-eager, 0.7% for eager-lazy, and 1.1% for lazy-lazy, the execution time is increased as well, offsetting any benefit. The exceptions are *kmeans* and *labyrinth*. The EDP for *kmeans* is an improvement over DVFS+CS using clock gating and is explained in the discussion in Section 5-4.4 For *labyrinth*, the average power increases by 118% for eager-eager and eager-lazy but whose execution time is decreased by 37%, resulting in a net loss in the power-performance domain. While the gating algorithm can save some energy in a hardware transactional memory system, it has the drawback of limiting the performance. For the serialization algorithm, the results are much the same (note that *kmeans* and *labyrinth* on the lazy-lazy system would not complete). Although there is a slight reduction (less than 1% on average) in the average power for most of the benchmarks, as with the clock gating method most of the reduction is offset by increased execution time.

The combined policies proposed in this paper provide between 21-30% improvement in EDP reduction relative to clock gating and serialization for eager conflict detection and 12-22% for lazy conflict detection. It is clear that for transactional programs with an abundance of contention, serialization and clock gating cannot improve the power and performance jointly and both the DVFS method and the contention prediction algorithm proposed in Sections 5-4.1 and 5-4.3 provide superior results. If the future of transactional memory is to increase the efficiency of parallel programming, then it can be expected that highly optimized programs like *cholesky* and *ocean* will not be the norm and programs are more likely to resemble some of the STAMP benchmarks. Regardless, to highlight the effect that the proposed methods

have on a range of transactional memory program behavior, synthetic benchmarks are needed.

**Synthetic Workloads**

This section provides an analysis of the power and performance of the different hardware transactional memory systems using synthetic benchmarks. Synthetic benchmarks [3] are miniature programs for use in early design evaluations. The advantage of synthetic benchmarks is that they can be used when the simulation time of real benchmarks is prohibitively long or for design space evaluation where no suitable benchmark exists, as is the case for this research. The benchmarks for this analysis are a parameterized form of workload synthesis derived using TransPlant [61].

For these experiments, the transactional granularity is scaled by powers of 2 beginning with 8 instructions and continuing to 128k instructions; the transaction stride, the distance between transactions, is equal to the transaction size so that the static number of transactional and sequential instructions remains equal. Memory accesses are modeled as circular arrays. On a per-thread basis, there is no reuse outside of the transaction that first references a specific location, ensuring that a single transaction in each thread can only interfere with a single transaction in another thread. For example in a program with n threads, TX1-A can interfere with $TX_2$-A, $TX_3$-A, …, and $TX_{n-1}$-A but never with $TX_n$-B, where n is the thread ID.

**Synthetic Workload Results**

Figure 5-6 shows the EDP of the synthetic benchmarks normalized to the base case for each example as transaction granularity increases. Immediately apparent is the abrupt shift in the trend at the 4k granularity. The reason for this relates to the average power of the transactional models. On the base system, the eager-lazy model has the

highest average power out of the three designs – peaking at 70W. At 4k, there is an abrupt drop in the average power for the two eager conflict detection schemes; eager-eager drops by 54% to 32W and eager-lazy drops by 60% to 41W while the average power for the lazy versioning scheme increases by an average of 6% until the granularity reaches 32k at which point it begins slowly decreasing. A breakdown of the transactional cycles, shown in Figure 5-7, is needed to further explain these phenomena.

The top graph in Figure 5-7 shows the relative execution time for the eager-eager system. Referring back to Figure 5-6, the reduction in EDP remains roughly flat until the transaction size reaches 4k and is a result of reduced power consumption from the conflict-aware scheduling policy; execution time remains mostly unchanged. At 4k, the benchmarks begin to spend more and more time in a NACKd state and the system is able to avoid aborting by stalling the processor, which itself reduces the average by 54% to 32W as pipelines become idle. DVFS and the conflict-aware scheduling policy are able to further reduce the power for all three schemes by an additional 60% and decrease the execution time by of the eager-conflict detection schemes by as much as 7%.

The eager-lazy behavior is shown in the middle graph of Figure 5-7. Again, until the transaction size reaches 4k, the average power remains roughly constant and is reflected by the almost constant EDP reduction. This is because the processors are able to spend almost 100% of the time performing useful work; there are no aborts and fewer stalls than the eager-eager model. The runtime is slightly increased by the proposed policies but is offset by moderate power reductions from the new scheduling

policy. At 4k, aborts and stalls completely overtake successful execution causing a 40% drop in the average power of the base system, from 67W to 41W. The proposed DVFS and conflict-aware scheduling policies are able to further reduce the power as well as the runtime, providing an additional 66% drop in power consumption and a 45% reduction in runtime.

The lazy-lazy cycle breakdown is shown in the bottom graph of Figure 5-7. There is more contention for lazy conflict detection than eager. For the 8- and 16-instruction transactions, the average power in the base system is lower than the other two by more than 30%. This is because the execution becomes serialized as transactions are waiting to commit, resulting in idle time for the processors. The DVFS scheme proposed in Section 5-4.1 is able to take full advantage of this fact, which is why there is a greater improvement in the power-performance domain for lazy-lazy. However, the potential for power and performance gains quickly diminishes as the transactions begin overlapping and aborts begin occurring. Beginning at 8k, the aborts become so persistent that the power manager essentially halts all but one processing element. Note that this is a different situation from the one in Section 5-4.4 where the scheduler was unnecessarily penalizing *raytrace* and *genome* but the result is the same – reduced power consumption relative to the baseline system but increased runtime, which increases the EDP.

## Related Work

The new work discussed in this paper is related to prior work on power management techniques. In [43], the authors show how processes can be mapped onto a variable number of processing elements while sleeping unused ones and guaranteeing some minimum performance threshold. Isci *et al.* [32] proposed managing

per-core voltage and frequency levels based on application behavior to manage total chip power. [78] proposed using linear programming to identify the optimal voltage and frequency levels for each core in a CMP to increase throughput and reduce EDP. Rangan *et al*. [62] show how threads can migrate between different PEs to achieve nearly the same power reduction as per-core DVFS while [41] propose an algorithm to improve fairness between co-executing threads. The drawback with all of these approaches is that they require online profiling of the runtime environment and computationally-intensive algorithms to meet their desired goals. The proposals outlined in this paper are less intrusive and achieve excellent results with minimal overhead.

There has been some recent research into the energy use of transactional memory for embedded and multiprocessor systems. Ferri recently proposed unifying the L1 and transactional cache in an embedded system and showed that using a small victim cache to reduce the pressure from sharing improved the energy-delay product [21]. Moreshet *et al.* [53] showed that hardware transactional memory systems can be more energy efficient than locks in the absence of contention. They then proposed a serialization mechanism for HTMs and showed that it lowered energy consumption for their microbenchmarks. However, their work relied on four non-contentious SPLASH-2 benchmarks and one in-house microbenchmark, making it difficult to draw any meaningful conclusions. Using an analytical model to estimate the additional power for an Alpha 21264, Sanyal *et al.* [68] proposed a technique for clock gating on an abort using TCC. Neither of these proposals exploit the feedback inherently available in transactional memory like the scheduler proposed by Yoo and Lee [88] who proposed an adaptive scheduler using parallelism feedback and showed speedups of almost 2x

for his selected benchmarks. While the energy reduction scheme proposed here has some similarities to previous work, it differs in two main regards. First, it abstracts the differences in the hardware, allowing for an almost direct comparison of power for different conflict detection and version management schemes. Secondly, the proposed method does not assume that contentious transactions should be serialized like [53] and is much less complicated than [68].

## Summary

Although there have been many proposed transactional memory designs, few have focused on the power-performances aspects of transactional memory. This research uses the SPLASH-2 and STAMP benchmark suites as well as synthetic workloads to analyze the power and energy for three different transactional systems: eager conflict/eager versioning, eager conflict/lazy versioning, and lazy conflict/lazy versioning and proposes two enhancements to HTM systems. The designs are kept simple by relying on power features available in modern processors and in proposed HTM designs. By targeting the idle periods in HTMs, the proposed optimizations reduce the average power and increase total throughput with minimal overhead.

To reduce system power and increase throughput when transactions are in a NACK state, a dynamic frequency and scaling system is proposed. By increasing the clock frequency of NACKing PEs and throttling the clock frequency of NACK'd PEs, the number of stall and abort cycles is reduced, increasing throughput. The PEs in low-power states serve to reduce or maintain the average system power. Together these effects serve to reduce the system EDP, or improve the power-performance of the system by 8% for eager-eager, 7% for eager-lazy, and 7% for lazy-lazy. To limit the number of aborts a program experiences and control power usage during these periods,

a new transaction scheduling policy is proposed that utilizes a transaction's current and past conflict density to determine whether a transaction should be preemptively stalled and its clock disabled. This technique provides an average reduction in the EDP of 17% for eager-eager, 17% for eager-lazy, and 10% for lazy-lazy. When applied together, the DVFS and scheduling policies provide a reduction in the EDP of 19% for eager-eager, 20% for eager-lazy, and 15% for lazy-lazy. More importantly, the benchmarks with greater contention (*labyrinth*) obtained even greater reductions – up to 76%. These results show the potential for manipulating clock frequencies for transactional memory for improved throughput while maintaining or reducing local and chip-wide power budgets and lay the foundation for future work in aggressive power management strategies for multithreaded workloads in the many-core era.

Table 5-1. Baseline Configuration

| | | Parameters |
|---|---|---|
| Core Model | Processing Elements | 2.4GHz, out-of-order, 4-issue, 65nm |
| | L1D Cache | 32kB, 4-way, 32B blocks, 2-cycle latency |
| | L2 Cache | 4MB, 8-way, 32B blocks, 9-cycle latency |
| | Off-chip memory | 240 cycle latency |
| | VDD | 0.6-1V (default of 1V) |
| Transactional Model | Conflict Detection | Eager and lazy |
| | Version Management | Eager and lazy |
| | Conflict Resolution | Requester/Committer wins with exponential backoff |
| | Conflict Granularity | 32B |
| | Primary Baseline | 50 |
| | Primary Variable | 9 |
| | Secondary Baseline | 12 |

Table 5-2. Frequency and Supply Voltage

| Freq (GHz) | 2.93 | 2.67 | 2.40 | 2.27 | 2.20 | 2.13 | 2.00 | 1.87 | 1.73 | 1.60 | 1.47 | 1.33 | 1.20 | 1.07 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Vdd (V) | 1.00 | 1.00 | 1.00 | 0.97 | 0.95 | 0.93 | 0.90 | 0.87 | 0.84 | 0.80 | 0.77 | 0.72 | 0.67 | 0.60 |

Table 5-3. Benchmark Parameters

| Benchmark | Abbreviation | Input | Benchmark | Abbreviation | Input |
|---|---|---|---|---|---|
| barnes | BN | 16K particles | labyrinth | LB | x32-y32-z3-n96 |
| bayes | BY | 1024 records | ocean-con | OC | 258x258 |
| cholesky | CH | tk15.O | ocean-non | ON | 66x66 |
| fluidanimate | FA | 35kMips | raytrace | RT | Teapot |
| fmm | FM | 16K particles | vacation | VA | 4096 tasks |
| genome | GN | g256 s16 n16384 | water-nsq | WN | 512 molecules |
| kmeans | KM | Random1000_12 | water-sp | WS | 512 molecules |

Table 5-4. Transactional- and Microarchitecture-Independent Characteristics From TransPlant

| Characteristic | Description |
|---|---|
| Threads | Total number of threads in the program |
| Homogeneity | All threads have the same characteristics |
| Tx Granularity | Number of instructions in a transaction |
| Tx Stride | Number of instructions between transactions |
| Read Set | Number of unique reads in a transaction |
| Write Set | Number of unique writes in a transaction |
| Shared Memory | Number of global memory accesses |
| Conflict Dist. | Distribution of global memory accesses |
| Tx Inst. Mix | Instruction mix of transactional section(s) |
| Sq Inst. Mix | Instruction mix of sequential section(s) |

Table 5-5. Performance Comparison ($nJ \cdot s^2$)

| | EE | | | EL | | | LL | | |
|---|---|---|---|---|---|---|---|---|---|
| | DVFS+CS | Gating | Serial | DVFS+CS | Gating | Serial | DVFS+CS | Gating | Serial |
| BN | 4.10E+10 | 4.65E+10 | 4.65E+10 | 4.24E+10 | 4.64E+10 | 4.65E+10 | 4.34E+10 | 4.65E+10 | 4.66E+10 |
| BY | 2.98E+11 | 3.58E+11 | 3.63E+11 | 2.53E+11 | 3.57E+11 | 3.62E+11 | 2.92E+11 | 3.50E+11 | 3.37E+11 |
| CH | 1.92E+08 | 2.09E+08 | 2.09E+08 | 1.92E+08 | 2.09E+08 | 2.09E+08 | 1.93E+08 | 2.09E+08 | 2.10E+08 |
| FA | 1.86E+10 | 1.98E+10 | 1.98E+10 | 1.86E+10 | 1.98E+10 | 1.98E+10 | 1.86E+10 | 1.99E+10 | 1.99E+10 |
| FM | 6.76E+10 | 7.15E+10 | 7.15E+10 | 6.76E+10 | 7.15E+10 | 7.15E+10 | 6.73E+10 | 7.15E+10 | 7.15E+10 |
| GN | 5.13E+04 | 5.63E+04 | 5.77E+04 | 5.13E+04 | 5.63E+04 | 5.77E+04 | 5.65E+04 | 5.72E+04 | 5.85E+04 |
| KM | 8.38E+08 | 9.22E+08 | 9.41E+08 | 8.38E+08 | 9.22E+08 | 9.41E+08 | 1.47E+09 | 9.03E+08 | |
| LB | 3.47E+07 | 2.85E+08 | 1.03E+09 | 3.29E+07 | 2.85E+08 | 1.03E+09 | 2.85E+07 | 1.12E+08 | |
| OC | 5.83E+09 | 6.23E+09 | 6.24E+09 | 5.83E+09 | 6.23E+09 | 6.24E+09 | 5.90E+09 | 6.23E+09 | 6.24E+09 |
| ON | 9.51E+06 | 9.76E+06 | 1.03E+07 | 9.51E+06 | 9.76E+06 | 1.03E+07 | 9.74E+06 | 9.94E+06 | 1.03E+07 |
| RT | 9.33E+07 | 9.95E+07 | 1.16E+08 | 9.33E+07 | 9.95E+07 | 1.16E+08 | 1.02E+08 | 1.15E+08 | 1.40E+08 |
| VA | 3.21E+07 | 4.03E+07 | 4.03E+07 | 3.21E+07 | 4.03E+07 | 4.03E+07 | 3.23E+07 | 4.03E+07 | 4.03E+07 |
| WN | 2.88E+08 | 3.10E+08 | 3.11E+08 | 2.88E+08 | 3.10E+08 | 3.11E+08 | 2.93E+08 | 3.11E+08 | 3.11E+08 |
| WS | 1.92E+08 | 2.03E+08 | 2.03E+08 | 1.92E+08 | 2.03E+08 | 2.03E+08 | 1.93E+08 | 2.03E+08 | 2.03E+08 |
| GEO | 4.86E+08 | 6.15E+08 | 6.88E+08 | 4.79E+08 | 6.15E+08 | 6.88E+08 | 5.08E+08 | 5.82E+08 | 6.56E+08 |

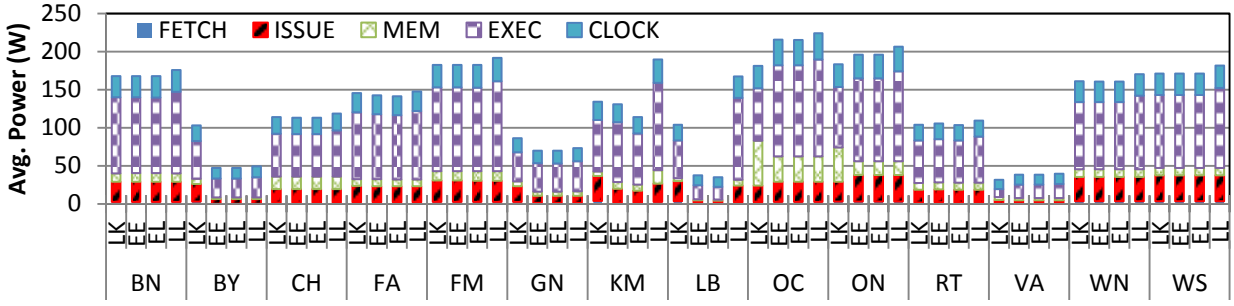

Figure 5-1. Benchmark Power (SPLASH-2 and STAMP)

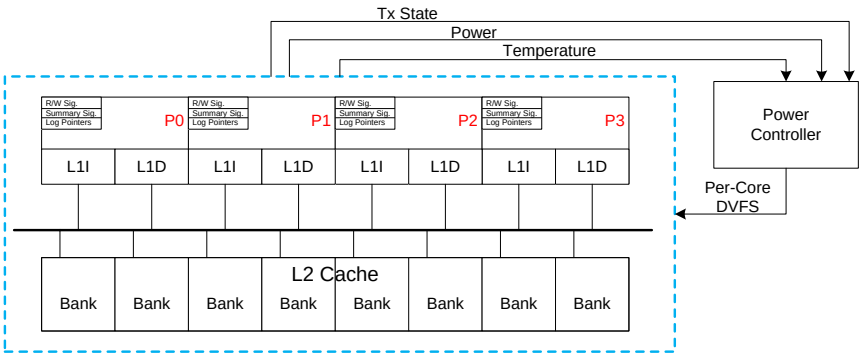


Figure 5-2. Baseline CMP Design

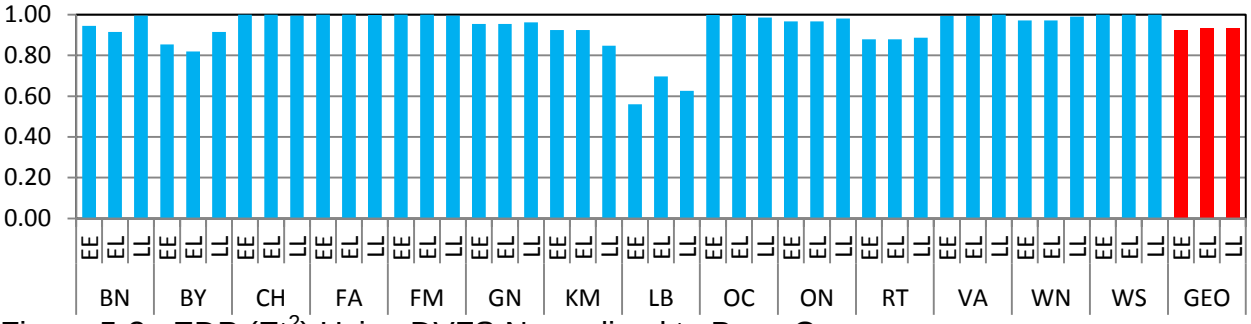


Figure 5-3. EDP ($Et^2$) Using DVFS Normalized to Base Case

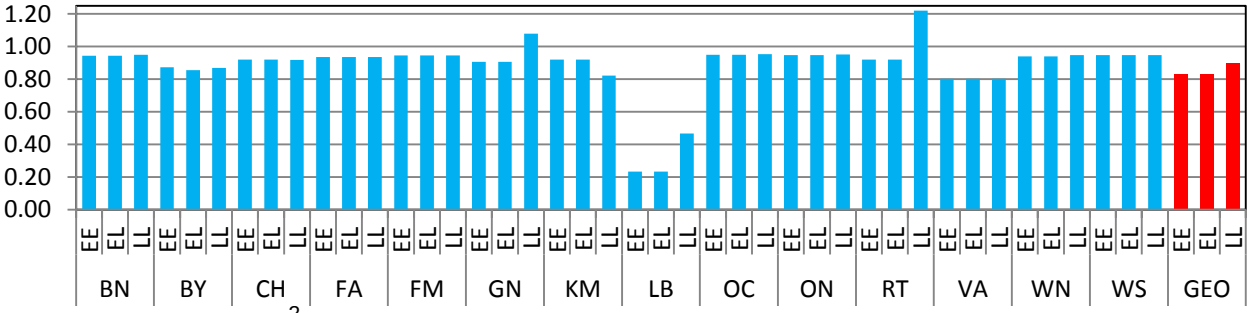


Figure 5-4. EDP ($Et^2$) Using Preemptive Stalling Normalized to Base Case

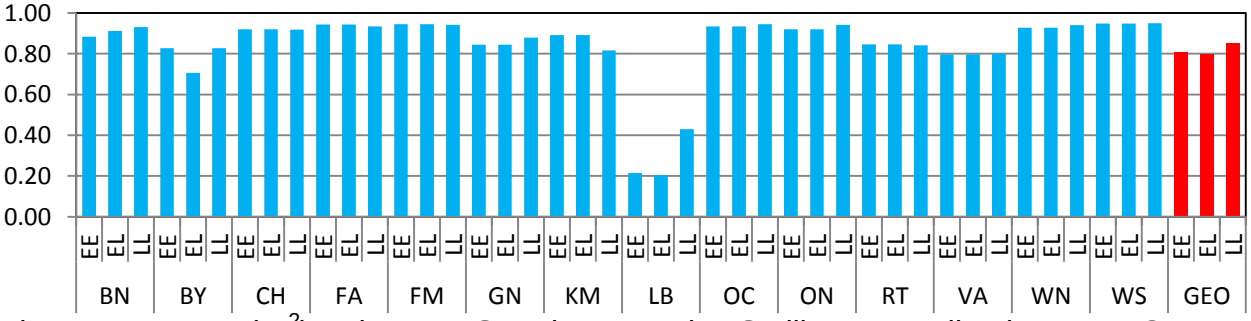


Figure 5-5. EDP ($Et^2$) Using DVFS and Preemptive Stalling Normalized to Base Case

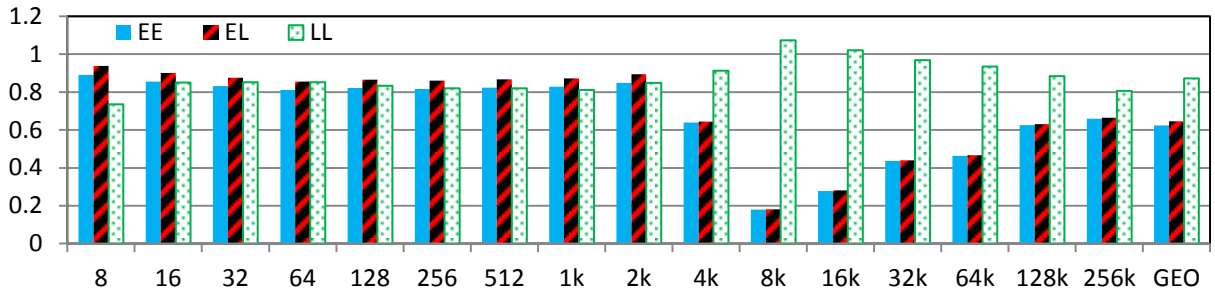


Figure 5-6. EDP ($Et^2$) Normalized to Base Case

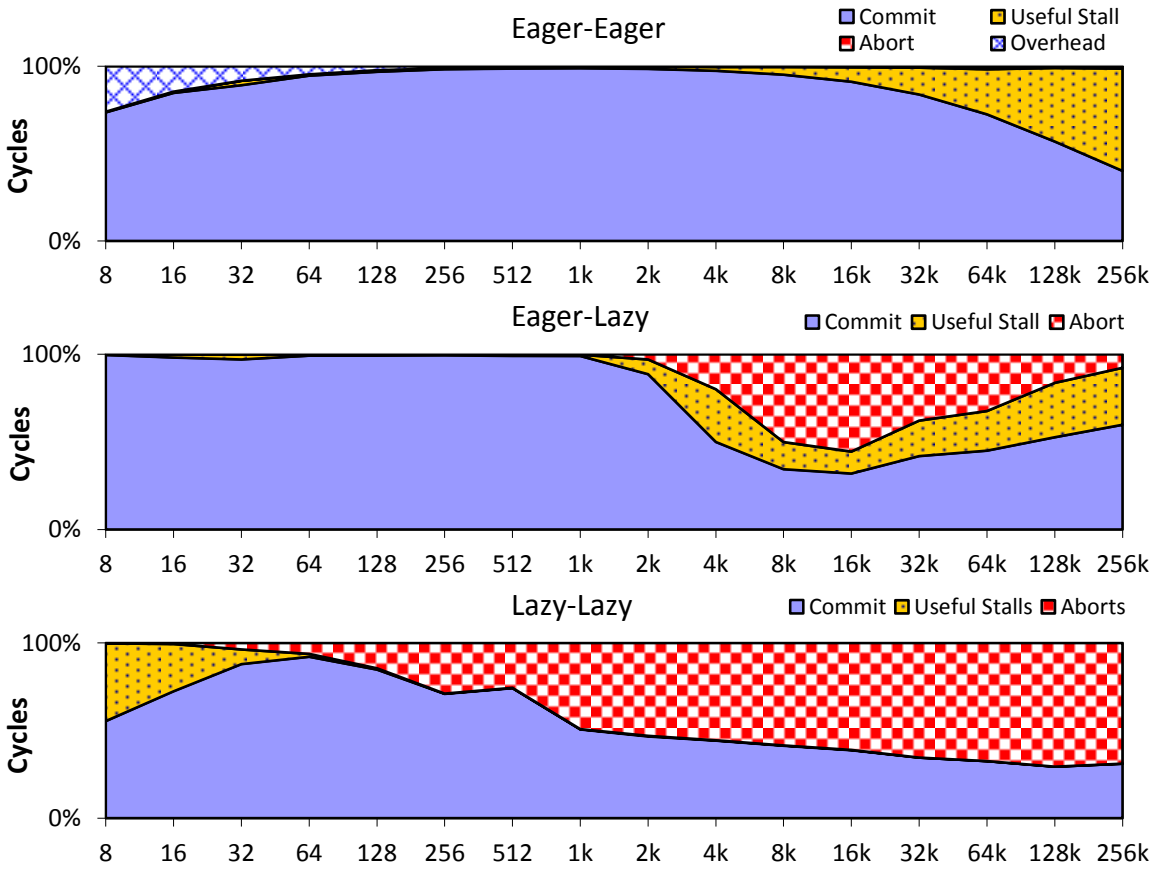


Figure 5-7. Relative Execution Time

LIST OF REFERENCES


[1] A. R. Alameldeen et al., "Evaluating Non-deterministic Multi-threaded Commercial Workloads," in *Workshop Computer Architecture Evaluation using Commercial Workloads*.

[2] R. H. Bell, L. Eeckhout, L. K. John, and K. De Bosschere, "Deconstructing and Improving Statistical Simulation in HLS," in *Workshop on Debunking, Duplicating, and Deconstructing*, 2004.

[3] R. H. Bell and L. K. John, "Improved Automatic Testcase Synthesis for Performance Model Validation," in *International Conference on Supercomputing*, 2005.

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

[5] M. V. Biesbrouck, L. Eeckhout, and B. Calder, "Considering All Starting Points for Simultaneous Multi-threading Simulation," in *International Symposium on Performance Analysis of Systems and Software*, 2006.

[6] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, pp. 422-426, July 1970.

[7] J. Bobba et al., "Performance Pathologies in Hardware Transactional Memory," in *International Symposium on Computer Architecture*, 2007.

[8] Boost C++ Libraries. [Online]. http://www.boost.org

[9] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-level Power Analysis and Optimization," in *International Symposium on Computer Architecture*, 2007.

[10] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk Enforcement of sequential Consistency," in *International Symposium on Computer Architecture*, 2007.

[11] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk Disambiguation of Speculative Threads in Multiprocessors," in *International Symposium on Computer Architecture*, 2006.

[12] D. Chiou et al., "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," in *International Symposium on Microarchitecture*, 2007.

[13] J.W. Chung et al., "The Common Case Transactional Behavior of Multithreaded Programs," in *International Symposium on High-Performance Computer Architecture*, 2006.

[14] I. Daubechies, *Ten Lectures on Wavelets*. Montelier, Vermont: Capital City Press, 1992.

[15] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *International Symposium on Distributed Computing*, 2006.

[16] L. Eeckhout, R. Bell, B. Stougie, K. De Bosschere, and L. John, "Improved Control Flow in Statistical Simulation for Accurate and Efficient Processor Design Studies," in *International Symposium on Computer Architecture*, 2004.

[17] L. Eeckhout and K. De Bosschere, "Hybrid Analytical-Statistical Modeling for Efficiently Exploring Architecture and Workload Design Spaces," in *International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[18] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere, "Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox," *IEEE Micro*, vol. 23, no. 5, pp. 26-38, 2003.

[19] 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*, vol. 5, 2003.

[20] M. Ekman and P. Stenstrom, "Performance and Power Impact of Issue-width in Chip-Multiprocessor Cores," in *ICPP*, Oct. 2003.

[21] C. Ferri, S. Wood, T. Moreshet, I. Bahar, and M. Herlihy, "Energy and Throughput Efficient Transactional Memory for Embedded Multicore Systems," in *International Conference on High-Performance Embedded Architectures and Compilers*, 2010.

[22] D. Genbrugge and L. Eeckhout, "Statistical Simulation of Chip Multiprocessors Running Multi-Program Workloads," in *International Conference on Computer Design*, 2007.

[23] D. Genbrugge, L. Eeckhout, and K. De Bosschere, "Accurate Memory Data Flow Modeling in Statistical Simulation," in *International Conference on Supercomputing* , 2006.

[24] L. Hammond et al., "Transactional Memory Coherence and Consistency," in *International Symposium on Computer Architecture*, 2005.

[25] T. Harris and K. Fraser, "Language Support for Lightweight Transactions," *SIGPLAN*, vol. 38, 2003.

[26] M. P. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data STructures," in *International Symposium on Computer Architecture*, 1993.

[27] C. Hsieh and M. Pedram, "Microprocessor Power Estimation using Profile-driven Program Synthesis," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 17, no. 11, pp. 1080-1089, 1998.

[28] C. Hsu and U. Kremer, "The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction," in *Conference on Programming Language Design and Implementation*, 2003.

[29] C. Hughes and T. Li, "Accelerating Multi-core Processor Performance Evaluation Using Automatic Multithreaded Workload Synthesis," in *IEEE International Symposium on Workload Characterization*, 2008.

[30] C. Hughes, J. Poe, A. Qouneh, and T. Li, "On The (Dis)similarity of Transactional Memory Workloads," in *IEEE International Symposium on Workload Characterization*, 2009.

[31] Intel C++ STM Compiler. [Online]. http://software.intel.com/

[32] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi, "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget," in *International Symposium on Microarchitecture*, 2006.

[33] T. Ishihara and H. Yasuura, "Voltage Scheduling Problem for Dynamically Variable Voltage Processors," in *International Symposium on Low Power Electronics and Design*, 1998.

[34] A. Jaleel, M. Mattina, and B. Jacob, "Last Level Cache Performance of Data Mining Workloads on a CMP – A Case Study of Parallel Bioinformatics Workloads," in *International Symposium on High-Performance Computer Architecture*, 2006.

[35] H. Jin, M. Frumkin, and J. Yan, "The OpenMP Implementation of NAS Parallel Benchmarks And Its Performance," Technical Report 1999.

[36] A. Joshi, L. Eeckhout, R. H. Bell, L. K. John, and K. De Bosschere, "Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks," in *IEEE International Symposium on Workload Characterization*, 2006.

[37] A.M. Joshi, L. Eeckhout, L.K. John, and C. Isen, "Automated Microprocessor Stressmark Generation," in *International Symposium on High Performance Computer Architecture*, 2008.

[38] A. Joshi et al., "Evaluating the Efficacy of Statistical Simulation for Design Space Exploration," in *International Symposium on Performance Analysis of Systems and Software*.

[39] J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood K.E. Moore, "LogTM: Log-based Transactional Memory," in *International Symposium on High-Performance Computer Architecture*, 2006.

[40] W. Kim, M. Gupta, G. -Y. Wei, and D. Brooks, "System Level Analysis of Fast, Per-Core DVFS Using On-Chip Switching Regulators," in *International Symposium on High-Performance Computer Architecture*, 2008.

[41] M. Kondo, H. Sasaki, and H. Nakamura, "Improving Fairness, Throughput, and Energy-Efficiency on a Chip Multiprocessor Through DVFS," *SIGARCH Computer ARchitecture News*, vol. 35, 2007.

[42] Y. Li, D. Brooks, Z. Hu, and K. Skadron, "Performance, Energy, and Thermal Considerations For SMT and CMP Architectures," in *International Symposium on High-Performance Computer Architecture*, Feb. 2005.

[43] J. Li and J. F. Martinez, "Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors," in *International Symposium on High-Performance Computer Architecture*, 2006.

[44] Man-Lap Li, R. Sasanka, S.V. Adve, Y. Chen, and E. Debes, "The ALP Benchmark Suite For Complex Multimedia Applications," in *IEEE International Symposium on Workload Characterization* , 2005.

[45] C.-K. Luk et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Conference on Programming Language Design and Implementation*, 2005.

[46] P. Macken, M. Degrauwe, M. V. Paemel, and H. Oguey, "A Voltage Reduction Technique For Digital Systems," in *IIEEE International Solid State Circuits Conference*, 1990, pp. 238-239.

[47] V. J. Marathe et al., "Lowering the Overhead of Non-blocking Software Transactional Memory," in *Workshop on Transactional Computing*, 2006.

[48] D. Marcalescu, "On the Use of Microarchitecure-Driven Dynamic Voltage Scaling," in *Workshop on Complexity-Effective Design*, 2000.

[49] R. McGowen et al., "Power and Temperature Control on a 90nm Itanium Family Processor," *Journal of Solid-State Circuits*, January 2006.

[50] C. C. Minh, K. Olukotun, C. Kozyrakis, and J. Chung, "STAMP: Stanford Transactional Applications for Multi-Processing," in *IEEE International Symposium on Workload Characterization*, 2008.

[51] C. C. Minh et al., "An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees," in *International Symposium on Computer Architecture*, 2007.

[52] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based Transactional Memory," in *International Symposium on High-Performance Computer Architecture*, 2006.

[53] T. Moreshet, R. I. Bahar, and M. Herlihy, "Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks," in *Workshop on Memory Performance Issues*, 2006.

[54] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "Minebench: A Benchmark Suite For Data Mining Workloads," in *IEEE International Symposium on Workload Characterization*, 2006.

[55] S. Nussbaum and J. E. Smith, "Modeling Superscalar Processors via Statistical Simulation," in *International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[56] S. Nussbaum and J. E. Smith, "Statistical Simulation of Symmetric Multiprocessor Systems," in *Annual Simulation Symposium*, 2002.

[57] M. Oskin, F. Chong, and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Design," in *International Symposium on Computer Architecture*, 2000.

[58] D.A. Penry et al., "Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multi-processors," in *International Symposium on High-Performance Computer Architecture*, 2006.

[59] C. Perfumo et al., "Dissecting Transactional Executions in Haskell," in *Workshop on Transactional Computing*, 2007.

[60] J. Poe, C. Cho, and T. Li, "Using Analytical Models to Efficiently Explore Hardware Transactional Memory and Multicore Co-Design," in *Computer Architecture and High Performance Computing*, 2008.

[61] J. Poe, C. Hughes, and T. Li, "TransPlant: A Parameterized Methodology For Generating Transactional Memory Workloads," in *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2009.

[62] K. K. Rangan, G. Wei, and D. Brooks, "Thread Motion: Fine-Grained Power Management for Multi-Core Systems," in *International Symposium on Computer Architecture*, 2009.

[63] C. H. Romesburg, *Cluster Analysis for Researchers*.: Lifetime Learning Publications, 1984.

[64] R. H. Saavedra and A. J. Smith, "Analysis of Benchmark Characteristics and Benchmark Performance Prediction," *ACM Transactions on Computer Systems*, vol. 14, no. 4, pp. 344-384, 1996.

[65] B. Saha, A. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "McRT-STM: A High Performance Software Transactional Memory System for a Multi-core Runtime," in *Symposium on Principles and Practice of Parallel Programmin*, 2006.

[66] B. Saha, A. Adl-Tabatabai, and Q. Jacobson, "Architectural Support for Software Transactional Memory," in *International Symposium on Microarchitecture*, 2006.

[67] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, "Implemetning Signatures for Transactional Memory," in *International Symposium on Microarchitecture*, 2009.

[68] S. Sanyal, S. Roy, A. Cristal, O. S. Unsal, and M. Valero, "Clock Gate on Abort: Towards Energy-efficient Hardware Transactional Memory," in *IEEE International Symposium on Parallel & Distributed Processing*, 2009.

[69] S. Sanyal et al., "Clock Gate on Abort: Towards Energy-efficient Hardware Transactional Memory," in *IEEE International Symposium on Parallel & Distributed Processing*, 2009.

[70] M.L. Scott, M.F. Spear, L. Dalessandro, and V.J. Marathe, "Delaunay Triangulation with Transactions and Barriers," in *IEEE International Symposium on Workload Characterization*, 2007.

[71] Semiconductor Industry Association (SIA). (2009) International Technology Roadmap for Semiconductors. [Online]. http://www.itrs.net/

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

[73] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[74] A. Shriraman et al., "An Integrated Hardware-Software Approach To Flexible Transactional Memory," in *International Symposium on Computer Architecture*, 2007.

[75] K. Skadron et al., "Temperature-Aware Microarchitecure," in *International Symposium on Computer Architecture*, 2003.

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

[77] D. Tarjan, S. Thoziyoor, and N. P. Jouppi, "CACTI 4.0," HP Labs, Technical Report 2006.

[78] R. Teodorescu and J. Torrellas, "Variation-Aware Application Scheduling and power Management for Chip Multiprocessors," in *International Symposium on Computer Architecture*, 2008.

[79] VTune. [Online]. http://www.intel.com/software/products/vtune

[80] Jyh-Ming Wang, Sung-Chuan Fang, and Wu-Shiung Fen, "New Efficient Designs for XOR and XNOR Functions on the Transistor Level," *IEEE Journal of Solid-State Circuits*, vol. 29, no. 7, 1994.

[81] J. Wang, S. Fang, and W. Fen, "New Efficient Designs for XOR and XNOR Functions on the Transistor Level," *IEEE Journal of Solid-State Circuits*, vol. 29, no. 7, 1994.

[82] I. Watson, C. Kirkham, and M. Lujan, "A Study of a Transactional Parallel Routing Algorithm," in *International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[83] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *International Symposium on Comptuer Architecture*, 1995.

[84] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," in *International Symposium on Computer Architecture*, 2003.

[85] L. Yen et al., "LogTM-SE: Decoupling Hardware Transactional Memory From Caches," in *International Symposium on High-Performance Computer Architecture*, Feb. 2007.

[86] L. Yen, S. C. Draper, and M. D. Hill, "Notary: Hardware Techniques to Enhance Signatures," in *International Symposium on Microarchitecture*, 2008.

[87] J.J. Yi et al., "Evaluating Benchmark Subsetting Approaches," in *IEEE International Symposium on Workload Characterization*, 2006.

[88] R. Yoo and H. S. Lee, "Adaptive Transaction Scheduling for Transactional Memory Systems," in *Symposium on Parallelism in Algorithms and Architecture*, 2008.

[89] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects," University of Virginia, CS-2003-05, 2003.

# BIOGRAPHICAL SKETCH

Clay Hughes was born in Enterprise, Alabama in 1977. He graduated Summa Cum Laude from Florida State University in 2005 with a Bachelor of Science in computer engineering where he was given the honor of being the 2006 Outstanding Graduate in Computer Engineering. He received his Master of Science degree from the University of Florida in 2007 from the Department of Electrical and Computer Engineering and completed his Doctor of Philosophy in 2010.