Accelerating Multi-core Processor Design Space Evaluation Using Automatic Multi-threaded Workload Synthesis

Clay Hughes and Tao Li
Intelligent Design of Efficient Architecture Lab(IDEAL)
http://www.ideal.ece.ufl.edu
Department of Electrical and Computer Engineering , University of Florida
cmhug@ufl.edu, taoli@ece.ufl.edu

Abstract

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 or 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 techniques, 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 multi-threaded 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, which can significantly accelerate architecture design evaluation of multi-core processors. We propose to construct synchronized statistical flow graphs that incorporate inter-thread synchronization and sharing behavior to capture the complex characteristics and interactions of multiple threads. Moreover, we develop thread-aware data reference models and wavelet-based branching models 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.

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 [1]. 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.

To accelerate multi-core design evaluation, we propose innovative techniques and methodologies for creating synthetic multi-threaded workloads with significantly reduced runtime. By applying techniques from statistical simulation to these elements, we generate accurate workload characterizations and produce a synthetic workload comprised of the dynamic execution features of the original multi-threaded program. We extend the concept of statistical flow graphs proposed by Eeckhout et al. [2] to include thread interactions. Moreover, we develop novel thread-aware data reference models and wavelet-based branching models to capture complex multi-threading memory access behavior and architectural independent dynamic branch characteristics. A

978-1-4244-2778-9/08/$25.00 ©2008 IEEE

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.

The rest of this paper is organized as follows: Section 2 provides a background on using workload synthesis to accelerate architecture design evaluation. Section 3 proposes a new method to produce synthetic multi-threaded workloads. Section 4 describes our framework implementation. Section 5 evaluates the accuracy and effectiveness of synthetic multi-threaded workloads. Section 6 summarizes related work. Section 7 concludes the paper.

2. 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, 4, 5] 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, 4, 5] 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 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, we propose a methodology to preserve this information and encode it into a synthetic representation of the original program.

```
#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 1. A sample multi-threaded program

3. 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, we include them together in the discussion below.

3.1 Multi-threaded Workload Representation

We extend the statistical flow graph (SFG) proposed in [2, 6] 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. We perform a basic block-level profiling of the original program to record a sequence of instructions within each basic block. If there is interaction with a threading library, we augment that basic block with additional information (such as the starting address of a spawn thread in the case of thread creation). We integrate the above information into synchronized statistical flow graphs (SSFG) which capture the statistical profile of both individual and interacted threads.

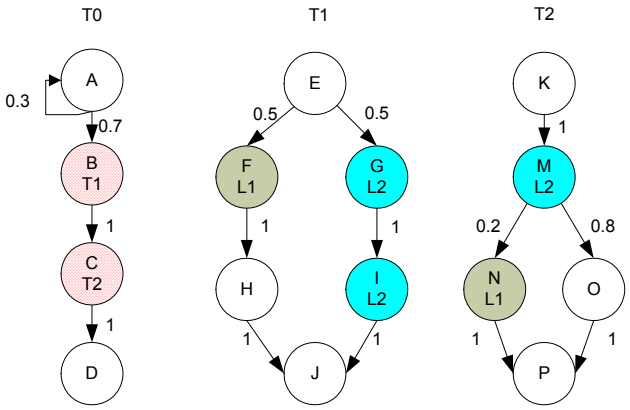

Figure 2. Sample SSFG. Edges are annotated to show transition probabilities and nodes are annotated to show control points (B and C in T0) and critical sections (F, G and I inT1 and N and M in T2) which are protected by locks L1 and L2.

Figure 2 illustrates an example of using the proposed synchronized statistical graphs to represent a program containing three separate threads. In Figure 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.

3.2 Statistical Flow Graph Reduction

Once a synchronized statistical flow graph is created for each thread, we apply the graph reduction factor method proposed by Eeckhout et al. [2] 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 our 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. We leave finding a heuristic that can be used to determine the optimal reduction factor as our future work.

3.3 Code Generation

Once the reduced statistical flow graphs are created, we use methods proposed for sequential workload synthesis [3, 5] 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 4.5.

4. Automatically Synthesizing Multi-threaded Workloads

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

4.1 The Front End

The front-end of our automatic multi-threaded workload synthesis framework is implemented using the Intel Pin tools [7], a dynamic instrumentation system capable of capturing the execution of an application by inserting customized code at key program locations. We use a disassembler 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 [8]. To help achieve this minimization, we make extensive use of the efficient data structures provided by the Boost library [9] to manage the graphs. While our

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.

4.2 Thread-aware Memory Reference Model

We propose to use a thread-aware memory reference model to capture original program's data reference locality. While prior work [10, 11] based their memory models on a program's cache and TLB miss rates, our 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. Our model distinguishes itself from previous stride-based memory models [3, 5] 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, we record the stride between successive references and store the result 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 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.

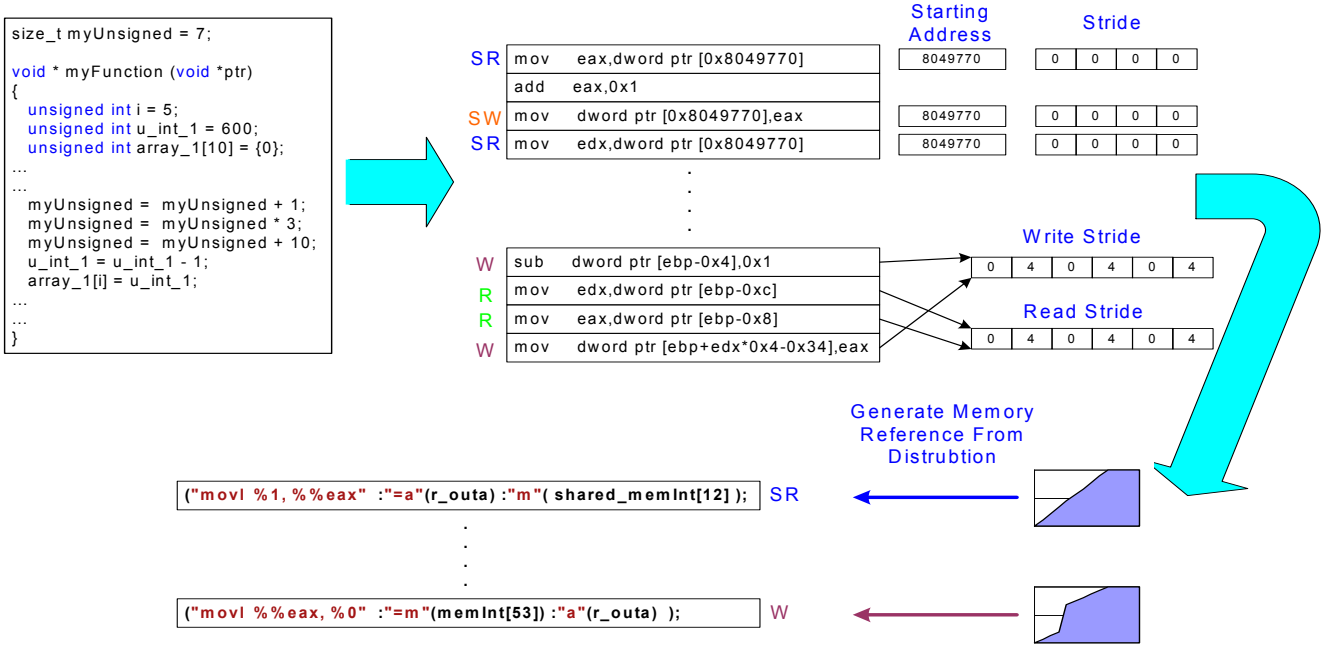

Figure 3. Thread-aware memory reference model

4.3 Flow Analysis

As mentioned in Section 4.1, to reduce perturbations in the system, which can influence the behavior of a multi-threaded program [8], 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. 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.

4.3.1 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 $Prob(N_n|N_{n-1})$ can be used to evaluate the transition probability for a given node, N_n.

4.3.2 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, we iterate 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.

4.4 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, [5] incorporates transition rates to filter out highly biased branches. To effectively capture workloads' complex branching patterns, we propose to profile the branch of each basic block and store its dynamic execution (e.g. taken or not-taken) as a bit vector. We found that a trace with length of 32 provides sufficient accuracy to capture branch dynamics of the experimented workloads. We treat each bit vector as a time series (e.g. 1 stands for taken and 0 represent not-taken) and apply wavelet analysis [12] 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. We use 16 wavelet coefficients 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, we use a representative pattern for all branches within the same cluster reducing the overhead of storing each block's branch pattern. Differing with prior work, our branch modeling technique cost-effectively captures complex branch dynamics and is independent of specific microarchitecture implementations.

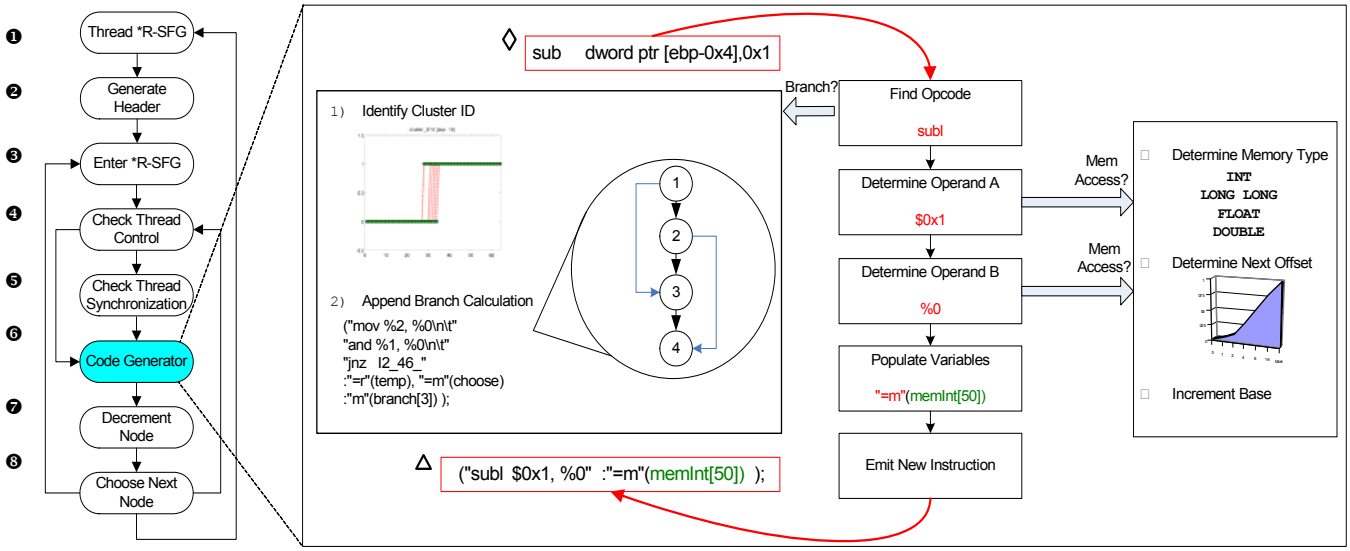

Figure 4. Control flow in code generator – *: Reduced SFG, ◊: Instruction from Pin, Δ: Synthesized instruction

4.5 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 4).

❶Choose the statistical flow graph of the next thread, beginning with thread zero (main thread).
❷Generate 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.
❸Begin 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.
❹If 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.
❺If 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.
❻Pass 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.
❼Decrement the node instance in the statistical flow graph.
❽A 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 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.

5. Evaluation

In this section, we examine the efficiency and accuracy of using synthetic multi-threaded workloads for multi-core performance evaluation. In addition, we contrast various workload and architecture characteristics between the synthetic and original multi-threaded benchmarks.

5.1 Experimental Setup

While the majority of research in workload synthesis and statistical modeling is performed in a simulation environment, the accuracy and efficiency of our proposed techniques were tested across three read-world hardware platforms. A summary of the system configurations for our test platforms are listed in Table 1. We are limited to Intel processor technology in our 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. We refer to these three machines as HT, Dual, and Quad in this paper.

Table 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. 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

In this study, we used nine SPLASH-2 benchmarks [22]: 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). We measured workload performance and execution characteristics using Intel's VTune analyzer [23]. Since multi-threaded workloads exhibit non-deterministic runtime behavior, we measured each workload (both original and synthetic versions) using multiple runs and reported average statistics.

5.2 Accuracy

To evaluate the accuracy of the proposed methodologies, we examine relative cross-platform speedup obtained from the synthetic benchmarks and compare 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 we are using multi-threaded programs, these 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 3 compares cross-platform speedup measured using both original and synthetic workloads with four threads. The cross-platform speedup is calculated using the formula:

$$Speedup\left(\frac{Quad}{Dual}\right)_{Original} = \frac{ExecutionTime(Dual)_{Original}}{ExecutionTime(Quad)_{Original}}$$

In addition, we compute the average absolute errors 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.

Table 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	
	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%)**	**(7.9%)**
Quad /HT	Original	2.87	1.8	1.96	3.03	2.8	3.45	2.93	2.28	3.92	
	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%)**	**(6.5%)**
Dual /HT	Original	1.27	1.02	1.55	1.82	2.28	2.12	1.7	1.24	1.44	
	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%)**	**(7.3%)**
	(Cross Platform Error)	**(6.9%)**	**(6.7%)**	**(5.6%)**	**(8.7%)**	**(8.2%)**	**(9.8%)**	**(3.8%)**	**(7.6%)**	**(7.6%)**	

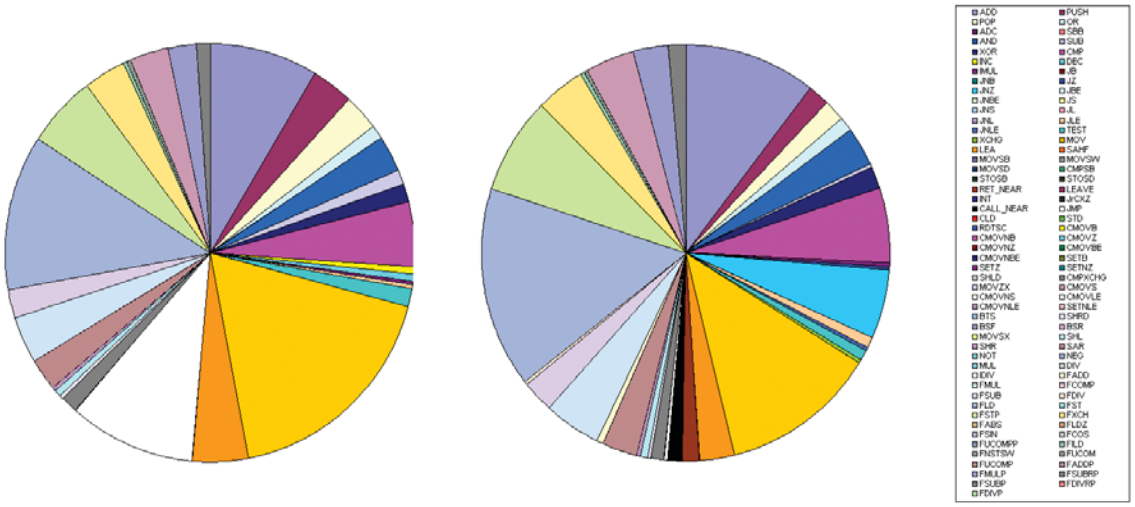

Figure 5. A comparison of instruction mix between synthetic (left) and original (right) FFT

5.3 Efficiency

Table 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

To evaluate the effectiveness of applying synthetic multi-threaded workloads to multi-core performance evaluation, we compare the execution runtime of the synthetic programs with that of the original applications. The results are presented as runtime reduction ratio in Table 4. In general, we observe more than an order of magnitude decrease in execution time. 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. We expect the technique to easily scale with large contemporary multi-threaded workloads and to produce synthetic programs with several orders of magnitude difference in runtime.

5.4 Workload Characteristics

We compare the inherent workload characteristics, 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 our techniques are capable of capturing thread activities and appropriately scaling down individual thread run time. Figure 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.

5.5 Microarchitecture Characteristics

We examined a variety of microarchitecture performance characteristics using 4-thread synthetic workloads. Each metric is compared with those of the original program. Figure 6 shows a comparison of CPI, L1 data cache and L2 cache hit rates, and branch prediction accuracy on the Pentium D system. We also performed microarchitecture characteristics analysis on the HT and Core 2 Quad machines and their error trends are similar. The maximum CPI discrepancy is 12% (Ocean-cont). Our 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, our 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.

We breakdown all references to the L2 cache based on the states of a cache block. The results on the Core 2 Quad platform are shown in Figure 7. A MESI based coherency protocol is used by the Core 2 Quad processors to maintain

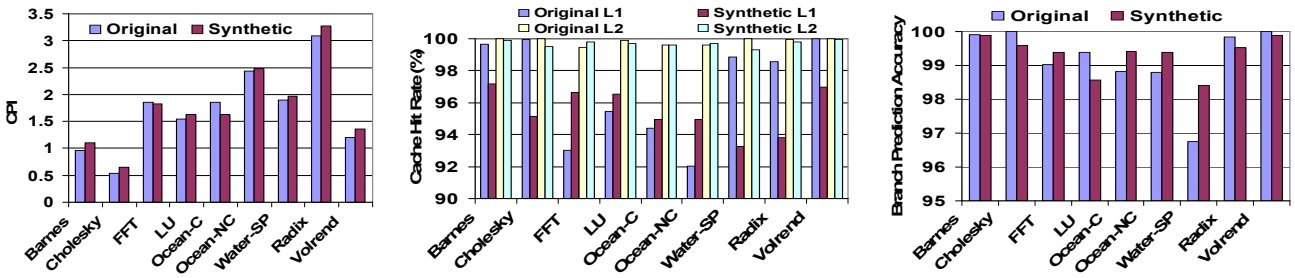

Figure 6. A comparison of CPI, cache hit rates, and branch prediction accuracy of the synthetic and original workloads

Table 5. Thread interaction comparison

		Barnes	*Cholesky*	*FFT*	*LU*	*Ocean-C*	*Ocean-N*	*Water-SP*	*Radix*	*Volrend*
Locked Operations Impact	Original	0.17%	1.27%	0.82%	0.25%	2.22%	2.62%	0.08%	0.64%	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	0.24	0.27	0.17	0.1	0.02	3.1	0.18	0.23	0.23
	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%**

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 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.

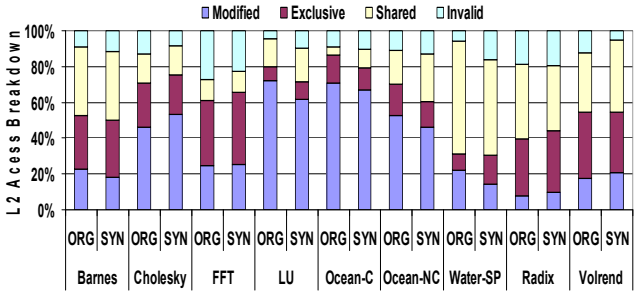

Figure 7. L2 Access breakdown by MESI states

5.6 Data Sharing and Thread Interaction

We use the advanced multi-core performance counters provided by the Core 2 Quad processors to analyze the impact of thread interaction on both the synthetic and original workloads. To be more specific, we examine VTune's modified data sharing ratio, locked operations impact, and data snoop ratio. 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 4 and 5, indicate that the synthetic significantly scales down the runtime while still faithfully preserving thread interaction.

5.7 Limitations

In this research, we use real hardware platforms 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 we can test. In our future work, we will perform additional studies using simulators and compare the results with those obtained using real hardware. Our 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 but we plan to include commercial and server multi-threaded workloads.

6. Related Work

SimPoint [13] and SMARTS [14] 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 [15] 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 [2, 10, 11, 16-21] 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 [2, 16]. To our knowledge, Nussbaum and Smith built the first statistical multiprocessor model [11] 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, [19] 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, we use workload characterization techniques to capture fine-grained, microarchitecture impendent thread interaction, memory accesses, and branch behavior. Our 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, we report both accuracy and efficiency of synthetic multi-threaded workloads across three real-world multi-threaded/multi-core processors. To our knowledge, this paper presents the first work to accurately and automatically synthesize multi-threaded workloads. [24] 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.

7. Conclusions

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, we propose to generate synchronized statistical flow graphs for multi-threaded programs. 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, we accurately capture and cost-effectively preserve 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. Our framework is modular and we expect to extend this framework to encompass a variety of threading languages and ISAs.

Acknowledgement

The authors would like to thank Dr. Robert Bell Jr. and the IBM Center for Advanced Studies (CAS) for their comment, feedback and support of this work. This research is partially supported by SRC 2008-HJ-1798 and an IBM Faculty Award.

References

[1] D. A. Penry, D. Fay, D. Hodgdon, R.Wells, G. Schelle, D. I. August and D. Connors, Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multi-processors, HPCA, 2006.

[2] L. Eeckhout, R. Bell Jr., B. Stougie, K. De Bosschere, and L. John, Improved Control Flow in Statistical Simulation for Accurate and Efficient Processor Design Studies, ISCA, 2004.

[3] R. H. Bell, Jr. and L. K. John, Improved Automatic Testcase Synthesis for Performance Model Validation, ICS, 2005.

[4] 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(11), pp. 1080-1089, 1998.

[5] A. Joshi, L. Eeckhout, R. H.Bell Jr., and L. K. John Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks, ISWC, 2006.

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

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapareddi, and K. Hazelwood, Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, PLDI, 2005.

[8] A. Alameldeen, *et al.,* Evaluating Non-deterministic Multi-threaded Commercial Workloads, Proceedings of the Computer Architecture Evaluation using Commercial Workloads, 2002.

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

[10] D. Genbrugge, L. Eeckhout, K. De Bosschere, Accurate Memory Data Flow Modeling in Statistical Simulation, ICS, 2006.

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

[12] I. Daubechies, Ten Lectures on Wavelets, Capital City Press, Montpelier, Vermont, 1992.

[13] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, Automatically Characterizing Large Scale Program Behavior, ASPLOS, 2002.

[14] R. E. Wunderlich, T. F. Wenisch, B. Falsafi and J. C. Hoe, SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling, ISCA, 2003.

[15] M. V. Biesbrouck, L. Eeckhout, and B. Calder, Considering All Starting Points for Simultaneous Multi-threading Simulation, ISPASS, 2006.

[16] S. Nussbaum and J.E. Smith, Modeling Superscalar Processors via Statistical Simulation, PACT, 2001.

[17] L. Eeckhout and K. De Bosschere, Hybrid Analytical-Statistical Modeling for Efficiently Exploring Architecture and Workload Design Spaces, PACT, 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, 23(5):26–38, 2003.

[19] D. Genbrugge and L. Eeckhout, Statistical Simulation of Chip Multiprocessors Running Multi-Program Workloads, ICCD, 2007.

[20] A. Joshi, J. J. Yi, R. H. Bell Jr., L. Eeckhout, L. K. John, and D. J. Lilja, Evaluating the Efficacy of Statistical Simulation for Design Space Exploration, ISPASS, 2006

[21] M. Oskin, F. Chong, M. Farrens, HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Design, ISCA, 2000.

[22] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, ISCA, 1995.

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

[24] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. Reinhart, E. Johnson, J. Keefe, and H. Angepat, FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators, MICRO, December 2007.