Automatic Generation of Miniaturized Synthetic Proxies for Target Applications to Efficiently Design Multicore Processors

Karthik Ganesan, *Member, IEEE,* and Lizy Kurian John, *Fellow, IEEE,*

Abstract—Prohibitive simulation time with pre-silicon design models and unavailability of proprietary target applications make microprocessor design very tedious. The framework proposed in this paper is the first attempt to automatically generate synthetic benchmark proxies for real world multithreaded applications. The framework includes metrics that characterize the behavior of the workloads in the shared caches, coherence logic, out-of-order cores, interconnection network and DRAM. The framework is evaluated by generating proxies for the workloads in the multithreaded PARSEC benchmark suite and validating their fidelity by comparing the microarchitecture dependent and independent metrics to that of the original workloads. The average error in IPC is 4.87% and maximum error is 10.8% for Raytrace in comparison to the original workloads. The average error in the power-per-cycle metric is 2.73% with a maximum of 5.5% when compared to original workloads. The representativeness of the proxies to that of the original workloads in terms of their sensitivity to design changes is evaluated by finding the correlation coefficient between the trends followed by the synthetic and the original for design changes in IPC, which is 0.92. A speedup of four to six orders of magnitude is achieved by using the synthetic proxies over the original workloads.

Index Terms—Computer Architecture, Synthetic Benchmarks, Multicore Systems, Workload Cloning and power modeling.

1 Introduction

When designing microprocessors, architects use simulation models at different levels of abstraction in combination with use case workloads to assess the performance and power consumption of a design. These simulation models are detailed cycle accurate models or RTL-level models that are at least a 1000X slower than real hardware. Running a complete application on these models incur prohibitive simulation time [1] making the design process very tedious. To reduce simulation time, sampling techniques like simulation points [2] and SMARTS [3] are well known and widely used. But, the problem with such sampling techniques is that most of them are restricted to phase behavior analysis and checkpointing of single-threaded applications and none of them can be directly used for sampling multithreaded applications. Though there has been some efforts towards extending such sampling techniques for multicore architectures as in the work by Biesbrouck et al [4], but it is all still in infancy due to the numerous combinations of thread samples that could occur. Another problem with such sampling techniques is that huge trace files for the particular dynamic execution interval have to be stored or they require the simulator to have the capability to fast-forward until it reaches the particular interval of execution that is of interest to the user. The problem with other techniques like benchmark subsetting [5] is that the results are still whole programs and are too big to be directly used with design models.

The previously mentioned simulation time problem is augmented with the unavailability of some of the real target applications due to being proprietary. For example, when a vendor is designing a system for defense applications or for military purposes, it is not possible to have these target applications in hand for performance analysis. In such cases, the architect will end up using the publicly available similar applications or the most generic benchmark suites. But, these proprietary target applications may have some unique characteristics that is not accounted for, and could result in the architects ending up with a non-optimal design.

In our recent research [6] [7] [8] [9], we have created a benchmark synthesis process, which consists of synthesizing a miniaturized proxy workload that possesses approximately the same performance and power characteristics as the original workload. The key idea behind such a benchmark synthesis framework is to identify the key characteristics of real world applications such as instruction mix, thread level parallelism, memory access behavior, branch predictability etc. that affect the performance and power consumption of a real program and create synthetic executable programs by controlling the values for these characteristics. Firstly, with such a framework, one can generate miniaturized synthetic proxies for large target (current and futuristic) applications enabling an architect to use them with slow low-level simulation models (e.g., RTL models in VHDL/Verilog) to tailor designs to these targeted applications. These synthetic benchmark clones can be distributed to architects and designers even if the original applications are proprietary and are not publicly available. These proxies

- *Karthik Ganesan is currently a senior member at Oracle Inc. and this work was done as a part of his Ph.D., which he received from the Department of Electrical and Computer Engineering, University of Texas at Austin in December 2011.*
- *Lizy Kurian John is a B. N. Gafford Professor in the Electrical and Computer Engineering department at the University of Texas at Austin.*

do not have any functional meaning and cannot be reverse engineered in any way to obtain any useful information about their original counterparts or their algorithms. The synthetic benchmarks that are provided are space efficient in terms of storage and do not require any special capability in a simulator as required by other simulation time reduction techniques [10] [2] [3].

1.1 Background

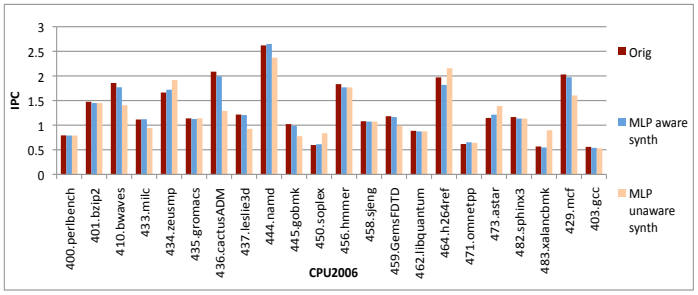
The idea of using statistical simulation to guide the process of design space exploration was introduced by Oskin et al. [11] and Nussbaum et al. [12]. The usage of Statistical Flow Graphs (SFG) was introduced by Eeckhout et al [13] in characterizing the control flow behavior of a program in terms of the execution frequency of basic blocks annotated with their mutual transition probabilities. The idea of synthesizing workloads based on profiles was introduced by Wong et al. [14] [15]. Synthesizing workload clones by populating embedded assembly instructions into loops was proposed by Bell and John [16], which was further extended by Joshi et al. [17]. The work by Bell and John included microarchitecture dependent metrics for workload profiling, but the work by Joshi et al. was the first effort to use microarchitecture independent metrics to profile the original workloads.

Previous work [17] [16] with respect to synthetic benchmark generation uses metrics categorized into control flow predictability, instruction mix, instruction level parallelism, data locality to clone single threaded applications. Ganesan et. al [6] [18] showed the importance of characterizing and using the Memory Level Parallelism (MLP) of the workloads along with other metrics to precisely model the execution behavior using synthetic benchmarks. In this previous recent work [6] of ours, we show that the synthetic proxies generated using our MLP-aware methodology have an error of only 2.8% in terms of Instruction Per Cycle (IPC) as compared to an error of 15.3% when using the previous MLP-unaware approaches for CPU2006. The Figure 2(a) shows the comparison of IPC between the original and synthetic workloads for CPU2006 for machine configuration as shown in Figure 1. We also evaluate their effectiveness in assessing the change in performance and power consumption for various microarchitecture design changes. For CPU2006, with synthetics limited to 1 million dynamic instructions, the average correlation coefficient for assessing design changes for IPC is 0.95 (0.98 for power-per-cycle). We achieved a speedup of up to six orders of magnitude in using the proxies for the CPU2006 workloads over the original applications. When cloning the futuristic workloads used in bio-implantable devices included in the ImplantBench suite [19], we have an average error of 2.9% in assessing the IPC as shown in Figure 2(b) for the machine configuration as shown in Figure 1. For ImplantBench, the correlation coefficient for assessing design changes is 0.94 (0.97 for power-per-cycle).

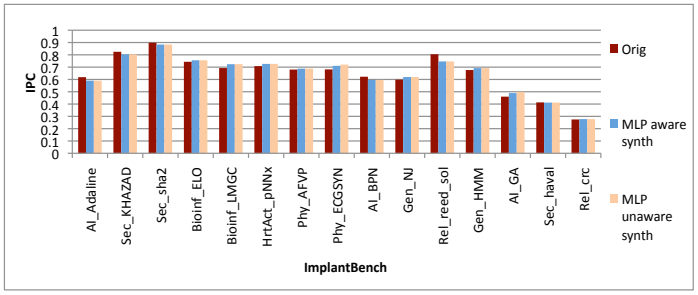
Automatic synthetic benchmark generation also has another application, which is to automatically search for stress benchmarks as demonstrated in our previous work [8] [7] [20] to aid in processor design. In Ganesan and John [7], power viruses

Configuration	Machine-A	Machine-B
Machine widths	Four wide out-of-order	one wide out-of-order
RUU size	128 entries	16 entries
LSQ size	64 entries	8 entries
Functional Units	4 Integer ALUs, 1 Integer Mul/Div, 4 FP ALUs and 1 FP Mul/Div	2 Integer ALUs, 1 Integer Mul/Div, 1 FP ALU and 1 FP Mul/Div
Memory	8B bus and 80 cycles access time	8B bus and 40 cycles access time
Branch Predictor	Bimod with table size 2048	2-level Gap predictor
IFQ size	32 entries	8 entries
L1 Data Cache	32 KB, 4 way, 32B line size, 2 cycle access time	16 KB, 2 way, 32B line size, 1 cycle access time
Unified L2 Cache	4MB, 4 way, 64B line size, 12 cycle access time	64 KB, 4 way, 64B line size, 6 cycle access time
L1 Instruction Cache	16 KB, block size 32 Bytes, 1 cycle access time	16 KB, block size 32 Bytes, 1 cycle access time

Fig. 1. Machine configurations used: Machine-A for SPEC CPU2006 and Machine-B for ImplantBench workloads



(a) SPEC CPU2006



(b) ImplantBench

Fig. 2. Comparison of IPC between the synthetic and the original workloads on single-core system configurations for Alpha ISA

are automatically generated for a given microarchitecture design for a single core system and are compared to other industry grade power viruses. We extended this work further to automatically generate power viruses [8] for multicore systems and compared their power consumption with other state-of-the-art power viruses and realistic user workloads like PARSEC and SPECjbb. In this prior work [8], we leverage the code generation framework with the help of a machine learning technique to search for stress benchmarks to maximize the power consumption for a given microarchitecture design of a multicore processor. On the other hand, to be able to clone parallel workloads that target multicore systems, the synthetic benchmark generation framework has to be more robust than what was previously proposed by us in [8]. Amongst the various applications that target Multicore systems, multithreaded applications are becoming increasingly common and these applications have a varied set of characteristics in terms of the

sharing patterns etc. that have an impact in the performance of the shared caches, the interconnection network, coherence logic and DRAM. In this paper, we propose an automatic synthetic benchmark generation framework, which is robust enough to clone multithreaded parallel applications to solve the problems due to large simulation time and unavailability of proprietary applications. The cloning methodology consists of a profiler that is used to get the characteristics of the long running original applications, and these characteristics are fed to the synthetic benchmark generation framework to generate proxies. These proxies are compared with the original applications based on both micro-architecture dependent and independent characteristics to evaluate their representativeness to their original counterparts.

To show the efficacy of the proposed synthetic benchmark generation in cloning multithreaded applications, proxies are generated for the benchmarks of the PARSEC suite [21]. The PARSEC benchmark suite is a collection of applications targeting shared memory multicore systems. Most of these applications are representative of the workloads that will be running on multicore desktop and server systems. The PARSEC suite also includes many emerging workloads that are expected to be more commonly used in the future. The benchmarks in the suite are not restricted to any single application domain, rather they are quite varied in terms of that usage. For example, PARSEC includes applications from the finance domain namely Blackscholes and Swaptions, that target option pricing using partial differential equations and a portfolio of swaptions respectively. The suite includes data mining applications like Streamcluster, Freqmine and Ferret targeting data clustering algorithms, itemset mining and content similarity search server respectively. The suite includes a workload Canneal that is used extensively in the chip design industry for optimizing routing cost of a chip design using simulated annealing. Compression algorithms like Dedup are also included. Image processing, video encoding and real time ray tracing algorithms are included, which are Vips, X264 and Raytrace respectively in the suite. A few applications from the physics domain like fluid dynamics for animation (Fluidanimate), body tracking of a person (Bodytrack) and simulation of face movements as in Facesim are also included.

In Section 2, we discuss the workload cloning methodology. First we elaborate on the metrics that are profiled out of the original applications and then we discuss how we use this information to automatically generate synthetic proxies. In Section 3, a characterization of the PARSEC workloads that is used for proxy generation is presented. In Section 4, we elaborate on the results in terms of the fidelity of the synthetic workloads in estimating the performance and power consumption of original workloads. Section 5 discusses the limitations and we summarize in Section 6.

2 Workload Cloning Methodology

The cloning methodology consists of two steps, (1) Characterize the original application based on a range of microarchitecture independent metrics as per our abstract workload model (2) Translate these characteristics with the help of a code generator to a real synthetic program, which will be a clone in terms of performance and power consumption to that of the original application. This final synthetic is compared with the original and the accuracies are reported for various machine configurations. Figure 3 shows the cloning methodology that is used.

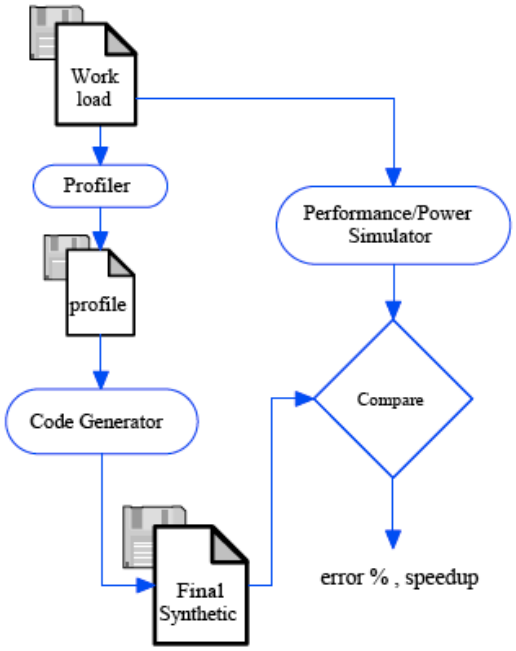


Fig. 3. Overall workload cloning methodology

In the process of workload cloning, the effectiveness of the synthetic benchmark generation framework lies in the efficacy of the abstract workload model that is formulated to characterize the original applications. The dimensions of this abstract workload space should be as much microarchitecture independent as possible to enable this framework to be able to generate synthetic benchmarks for different types of microarchitectures for the purposes of design space exploration. These dimensions should also be robust enough to be able to vary the execution behavior of the generated workload in every part of a multicore system. In earlier approaches for synthetic benchmark generation at core-level for uniprocessors, researchers came up with metrics to characterize the execution behavior of programs on single core processors [17] [22] [23] [16]. In this research, we come up with similar metrics for the generation of system-level synthetics and for multicore systems. We first begin by explaining the intuition behind the design of this abstract workload space in terms of our memory access model, branching model and shared data access patterns.

2.1 Data Sharing Patterns

Investigation in previous research [24][25][26][27][28] about the communication characteristics of the parallel applications has showed that there are four significant data sharing patterns that happen, namely,

1) **Producer-consumer sharing pattern**: One or more producer threads write to a shared data item and one

No.	Metric	Category
1	Average basic block size	Control flow predictability
2	Branch transition rate	
3	INT ALU proportion	Instruction mix
4	INT MUL proportion	
5	INT DIV proportion	
6	FP ADD proportion	
7	FP MUL proportion	
8	FP DIV proportion	
9	FP MOV proportion	
10	FP SQRT proportion	
11	LOAD proportion	
12	STORE proportion	
13	Dependency distance distribution per instruction type	Instruction level parallelism
14	Private stride value per static load/store	Data locality
15	Data Footprint of the workload	
16	Mean and standard deviation of the MLP	Memory Level Parallelism (MLP)
17	MLP frequency	
18	Number of threads	Thread level parallelism
19	Thread class and processor assignment	Shared data access pattern and communication characteristics
20	Percentage loads to private data	
21	Percentage loads to read-only data	
22	Percentage migratory loads	
23	Percentage consumer loads	
24	Percentage irregular loads	
25	Percentage stores to private data	
26	Percentage producer stores	
27	Percentage irregular stores	
28	Shared stride value per static load/store	
29	Data distribution based on sharing patterns	
30	Lock/unlock pair frequency	Synchronization Characteristics
31	Conflict density using mutex objects	
32	Lock to unlock distance	

Fig. 4. List of metrics to characterize the execution behavior of workloads that significantly affect the performance and power consumption

or more consumers read it. This kind of sharing pattern can be observed in the SPLASH-2 benchmark *Ocean*.

2) **Read-only sharing pattern**: This pattern occurs when the shared data is constantly being read and is not updated. SPLASH-2 benchmark *Raytrace* is a good example exhibiting this kind of a behavior.
3) **Migratory sharing pattern**: This pattern occurs when a processor reads and writes to a shared data item within a short period of time and this behavior is repeated by many processors. A good example of this behavior will be a global counter that is incremented by many processors.
4) **Irregular sharing**: There is not any regular pattern into which this access behavior can be classified into. A good example will be a global task queue, which can be enqueued or dequeued by any processor which does not follow a particular order.

Though the above said patterns are the most commonly occurring sharing patterns, parallel workloads may have a combination of one or more of the aforementioned patterns. In our framework, we use a generic memory access model, which when parameterized accordingly, can yield any combination of the above said sharing patterns.

2.2 Stride Based Memory Access Behavior

Capturing the data access pattern of the workload is critical to replay the performance of the workload using a synthetic benchmark. The data access pattern of a benchmark affects the amount of locality that could be captured at various levels of the memory hierarchy. Though locality is a global metric characterizing the memory behavior of the whole program, our memory access model is mainly based on a 'stride' based access pattern [17] in terms of static loads and stores in the code. When profiling a modern workload, one can observe that each of the static loads/stores access the memory like in an arithmetic progression, where the difference between the addresses of two successive accesses is called the stride. It is known that the memory access pattern of most of the SPEC CPU2006 workloads can be safely approximated to be following a few dominant stride values as in our previous work [6]. In our abstract workload model, the stride values of the memory accesses to the private and shared data are handled separately.

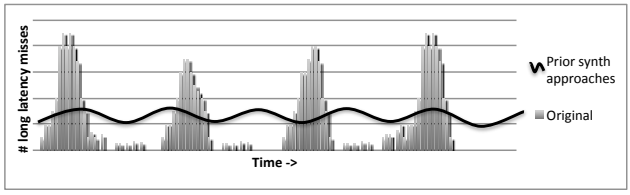


Fig. 5. Comparison of the MLP behavior of synthetics generated by previous approaches to that of a real single-threaded workload

2.3 Model for the Memory Level Parallelism

Even for single-core systems, the previous synthetic benchmark generation efforts [17] [16] suffer from a major limitation. Their methodologies characterize the memory access, control flow and the instruction level parallelism information of the workload, but do not characterize or use the burstiness of memory accesses, which is the Memory Level Parallelism (MLP) information. As shown in Figure 5, the execution behavior and performance of the original and synthetic benchmarks may vary significantly even when they have the same miss rates in the caches. The original workloads can have a set of bursty long-latency loads in one time interval of execution and none of them at all for another interval of execution. These two behaviors will have very different execution times for a workload.

2.4 Transition Rate Based Branch Behavior

The branch predictability of the benchmark can be captured independent of the microarchitecture by using the branch transition rate [29]. The branch transition rate captures the information about how quickly a branch transitions between taken and not-taken paths. A branch with a lower transition rate is easier to predict as it sides towards taken or not-taken for a given period of time and rather a branch with a higher transition rate is harder to predict. First, the branches that have very low transition rates, can be generated as always taken or always not taken as they are easily predictable. The rest of the branches in the synthetic need to match the specified

distribution of transition rate, which is further explained in the next Subsection.

2.5 Dimensions of the Abstract Workload Model

Our workload space consists of a set of 25 dimensions falling under the categories of control flow predictability, instruction mix, instruction level parallelism, data locality, memory level parallelism, shared access patterns, synchronization as shown in Figure 4. Further in this Subsection, each of these dimensions or what we call as the 'knobs' of our workload generator in this framework are explained:

1) **Number of threads**: The number of threads knob controls the amount of thread level parallelism of the synthetic workload.
2) **Thread class and processor assignment**: This knob controls how the threads are mapped to different processors in the system. There are many thread classes to which each thread gets assigned. The threads in the same class share the same characteristics.
3) **Number of basic blocks**: The number of basic blocks in the program combined with the basic block size determines the instruction footprint of the application. The number of basic blocks present in the program has a significant impact on the usage of the instruction cache affecting the performance and power consumption based on the Instruction cache missrates.
4) **Shared memory access stride values**: This knob can be used to provide a set of stride values that should be followed by the loads and the stores that access shared data.
5) **Private memory access stride values**: This knob can be used to provide a set of stride values that should be followed by the loads and the stores that access private data.
6) **Data footprint**: This knob controls the data footprint of the synthetic. The data footprint of the application controls the number of cache lines that will be touched by the different static loads and stores. Also, it has a direct impact on the power consumption of the data caches.
7) **Memory Level Parallelism (MLP)**: This knob controls the amount of Memory Level Parallelism (MLP) in the workload, which is defined as the number of memory operations that can happen in parallel and is typically used to refer to the number of outstanding cache misses at the last level of the cache. The number of memory operations that can occur in parallel is controlled by introducing dependency between memory operations. The memory level parallelism of a workload also affects the power consumption due to its impact on the DRAM power and also the pipeline throughput.
8) **MLP frequency**: Though the MLP knob controls the burstiness of the memory accesses, one needs one more knob to control how frequently these bursty behaviors happen.
9) **Basic block size and execution frequency**: Basic block size refers to the average and standard deviation of number of instructions in a basic block in the generated embedded assembly based synthetic code. Execution frequency of basic block is used when detailed instruction pattern information has to be reproduced in the synthetic while cloning.
10) **Branch predictability**: The branch predictability of a workload is an important characteristic that also affects the overall throughput of the pipeline. When a branch is mispredicted, the pipeline has to be flushed and this results in a reduced activity in the pipeline.
11) **Instruction mix**: The Instruction mix is decided based on the proportions of each of the instruction types INT ALU, INT MUL, INT DIV, FP ADD, FP MUL, FP DIV, FP MOV and FP SQRT. Since the code generator generates embedded assembly, we have direct control over the instruction mix of the generated workload. Some restrictions are placed on the instruction mix by writing rules in the code generator like a minimum number of INT ALU instructions should be present if there are any memory operations in the code to be able to perform the address calculation for these memory operations.
12) **Register dependency distance**: This knob refers to the average number of instructions between the producer and consumer instruction for a register data. The proportion of instructions that have an immediate operand is also used along with this distribution. This distribution is binned at a granularity of 1, 2, ... 20, 20-100 and greater than 100. If the register dependency distance is high, the Instruction Level Parallelism (ILP) in the synthetic is high resulting in a high activity factor in the pipeline of the core.
13) **Random seed**: This knob controls the random seed that is used as an input to the statistical code generator, which will generate different code for the same values for all the other knobs. It mostly affects the alignment of the code or the order in which the instructions are arranged.
14) **Percentage loads to private data**: This knob refers to the proportion of load accesses that are to the private data and the rest of the memory accesses are directed to shared data.
15) **Percentage loads to read-only data**: This knob refers to the percentage of loads that access read-only data. Since this part of the data does not have any writes, they do not cause any invalidation traffic in the interconnection network. The main traffic that will be generated by this kind of data will be capacity misses and data refills from other caches.
16) **Percentage migratory loads**: This knob refers to the percentage of loads that are coupled with stores to produce a migratory sharing pattern. We cannot separately use a knob for migratory store percentage as it is co-dependent on this knob. This migratory sharing pattern can create huge amounts of traffic when a coherence protocol like MESI is used where there is not a specific state for a thread to own the data.
17) **Percentage consumer loads**: This knob refers to the percentage of loads that access the producer consumer

data. The stores are configured to write to this producer consumer data and some loads are configured to read from them to reproduce the producer-consumer sharing pattern.

18) **Percentage irregular loads**: This knob refers to the percentage of loads that fall into the irregular sharing pattern category and they just access the irregular data pool based on the shared strides specified.
19) **Percentage stores to private data**: This knob controls what proportion of stores access the private data and the rest of the memory accesses are directed to shared data.
20) **Percentage producer stores**: This knob refers to the percentage of stores that write to the producer consumer data to replay the producer-consumer sharing pattern.
21) **Percentage irregular stores**: This knob refers to the percentage of stores that fall into the irregular sharing pattern category and they write to the irregular data pool.
22) **Data pool distribution based on sharing patterns**: This knob controls how the spatial data is divided in terms of the different sharing pattern access pools. It determines the number of arrays that are assigned to private, read-only, migratory, producer-consumer and the irregular access pools for the synthetic.
23) **Number of lock/unlock pairs**: This knob refers to the number of lock/unlock pairs present in the code for every million instructions. This knob is very important to make synthetics that are representative of multithreaded programs that have threads synchronizing with each other using locks.
24) **Number of mutex objects**: This knob controls the number of objects that will be used by the locks in the different threads. It controls the conflict density between the threads when trying to acquire a lock. When the number of mutex objects is increased, the conflict density gets reduced and in turn the workload executes much more efficiently resulting in a higher per-thread IPC.
25) **Number of Instructions between lock and unlock**: This knob controls the number of instructions in the critical section of the workload. The bigger the critical section, the longer will be the wait to acquire locks by threads as it takes longer to finish executing all the instructions in the critical section and release a lock.

2.6 Code Generation

This section elaborates on how the final code generation happens based on the knob settings given in terms of the abstract workload parameters. Figure 6 shows an overview of code generation. The generated code consists of the main function and a function for each thread that is spawned from the main function using the pthread_create() system call. The required amount of shared data is declared and allocated in the main function as a set of integer/floating point arrays and the pointers to these arrays are available to each of the threads. The private data that is supposed to be used by every thread is declared and allocated within the function for each thread. Each of the threads also bind themselves with the processor number specified when the code was generated based on the thread class and processor assignment knob. A barrier synchronization is used to synchronize all the threads after they finish their respective system calls for allocating their private data arrays and binding themselves to the assigned processor.

The body of each thread consists of two inner loops filled with embedded assembly and one outer loop encompassing these inner loops. As previously mentioned, our memory model is a stride based access model, where the loads and stores in the generated synthetic access the elements of the private/shared arrays, each static load/store with a constant stride. The address calculation for the next access of each load/store is done by using other ALU instructions in the generated code for each of the array pointers by using the assigned stride value. When the specified data footprint is covered, the pointers that are used are reset to the beginning of the array. This pointer reset is done outside the inner loops and inside the encompassing outer loop enabling us to control the data footprint with the number of iterations of the inner loop and the number of dynamic instructions with the number of iterations of the outer loop. The embedded assembly contents of the two inner loops are the same except the MLP behavior of the second loop is different from that of the first loop based on the MLP frequency. If an original workload has high MLP, but if it occurs at a very low frequency, one will need at least two loops to match that behavior.

Out of the total number of registers in the ISA, a set of registers are allocated to hold the base addresses of these allocated memory arrays and another set of registers are used to implement the predictability of the branches. The structure of our inner most loop is similar to that of the one proposed by Bell, et al. [16], but with an improved memory access, branching and ILP models. The required branch predictability or the control flow behavior in the synthetic is achieved by grouping branches into pools with each pool assigned to a constantly incremented register and a modulo operation on the register is used to decide if that branch is taken or not taken. The only information that is required to generate the main function is the biggest shared data footprint amongst the different threads to be able to allocate the shared arrays as shown in Figure 6. The following steps are followed to generate the code for every thread based on the corresponding knob settings for each:

1) Generate the code to allocate the required amount of memory for private data accesses based on the percent private accesses, proportion of memory operations in instruction mix and the data footprint. The number of 1-D shared arrays are further subdivided into pools for each of the sharing patterns based on the spatial shared data access information.
2) Generate the processor_bind() system call using the assigned processor number and then a barrier synchronization system call is generated as shown in Figure 6.
3) Generate the code for outer-loop based on the dynamic number of instructions desired taking into account the average basic block size and the number of basic blocks.
4) Fix the code spine for the first inner loop based on a

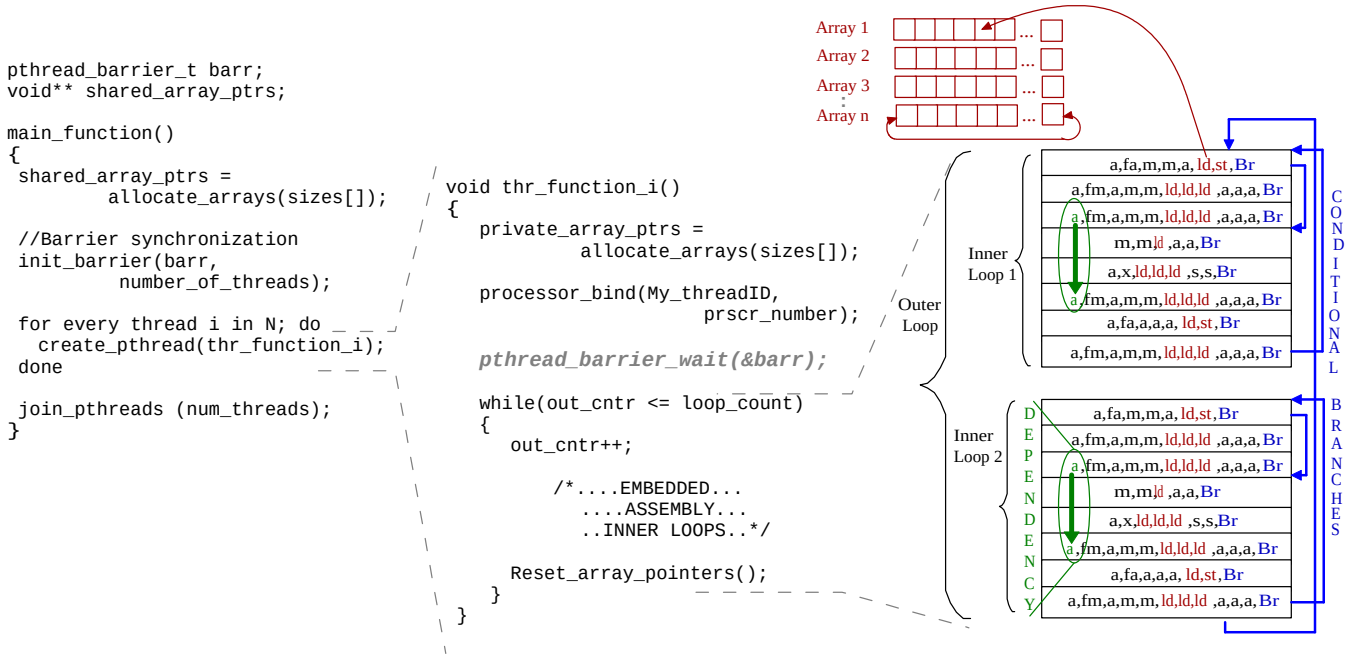


Fig. 6. Multithreaded synthetic workload generation

fixed number of basic blocks and the average basic block size knob.

5) For each of the basic block in the first inner loop, configure the instruction type of each instruction by stochastically choosing from the instruction mix information. Based on the percentage migratory loads, a few load instructions are made to be followed by a store and these couple load-stores are forced to access the same address.
6) The basic blocks are bound together by using conditional branches at the end of each of the basic block. The number of branch groups and the modulo operation are fixed based on the required average branch predictability. The modulo operation for each of the branch groups are generated at the beginning of the inner loop based on the loop count and a register is set/unset to decide if those branches for this particular group are going to be taken or not taken for this loop iteration. Branches are generated to fall through or take the target to another basic block based on their assigned register value.
7) Using the average dependency distance knob, each of the operands of every instruction is assigned with a previous producer instruction. Some rules are used to check the compatibility between producer and the consumer in terms of the data that is produced by the producer instruction and that consumed by the consumer. If two instructions are found to be not compatible, the dependency distance is incremented or decremented until a matching producer is found for every instruction. The memory level parallelism information is also used to assign load-load dependencies in this process.
8) Based on the percent private accesses knob, each of the memory operations are classified into the ones that access shared data and the ones that access private data. Based on the stride value of the corresponding memory operation (shared or private and based on the assigned bin), their corresponding address calculation instructions are given the stride values as immediate operands. If a load, store is classified to access shared data, the sharing pattern pool that they should access is determined by rolling a dice and using the shared data access pattern information.
9) Register assignment happens by first assigning the destination registers in a round robin fashion. The source register for each operand of an instruction is assigned as the destination register of the producer instruction based on the corresponding dependency assignment.
10) The loop counters for the inner loops are set based on the specified data footprint and the compare instructions for loop termination are generated by choosing an integer ALU instruction in the code.
11) The second inner loop is also generated which is a copy of the first loop, except with different MLP behavior by altering the load-load dependencies.
12) Outside each of these inner loops, the memory base registers are reset to the first element of the memory arrays to enable temporal locality for the next loop or the next iteration of the outer wrapper loop.
13) Based on the number of locks, instructions between lock/unlock pairs provided as input to the code generator, the pthread_lock and pthread_unlock function calls are inserted in between the embedded assembly instructions. The mutex object to be used is determined by rolling a dice and choosing from the number of mutex objects specified.

During the synthesis of the workload, the desired MLP in

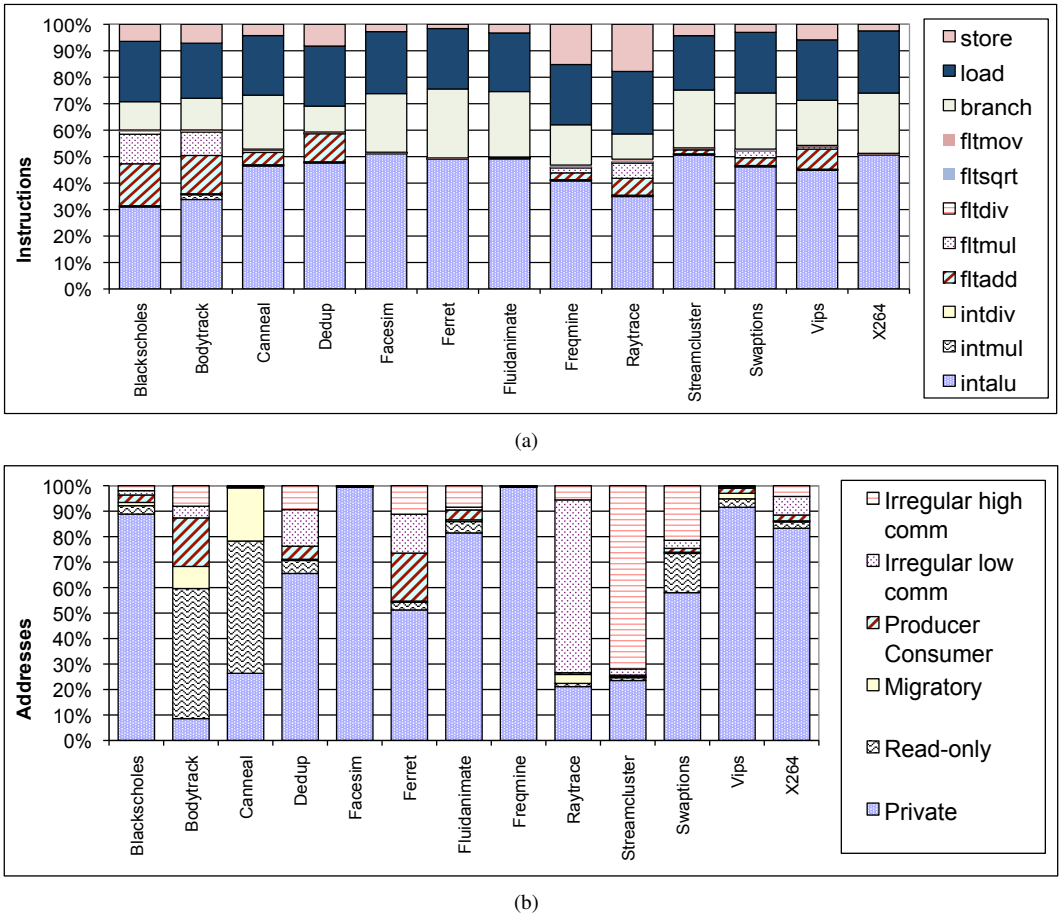


Fig. 7. (a) Instruction mix distribution of various PARSEC workloads (b) Spatial distribution of the accessed memory addresses into sharing patterns of various PARSEC workloads

the synthetic is incorporated by having control over the load-load dependencies in the code. When a load is dependent on another load, these dependent load instructions cannot be sent out to the memory at the same time and thus controlling the amount of MLP in the synthetic.

3 Benchmark Characterization

This subsection elaborates on how each of the different metrics of the abstract workload model are captured and also provide some of the characterized data for the PARSEC workloads. For PARSEC workloads, we use the input set provided for detailed microarchitectural simulations called simsmall. The full system simulator Windriver Simics is used along with the processor, memory and interconnection network simulation model called GEMS [30] from the University of Wisconsin-Madison for profiling the workloads. An instruction trace and a memory access address trace are captured and post-processed on the fly to record most of the significant characteristics. The instruction mix, register dependency distance distribution and the various synchronization characteristics are recorded based on the instruction trace. Figure 7(a) shows the distribution of the instruction into various categories of instruction types for the PARSEC workloads. It can be noted that most of the PARSEC applications are quite compute intensive in terms of the integer operations. Only a few workloads have a considerable amount of floating point operations namely Blackscholes, Bodytrack and Canneal. Most of the Parsec workloads have 20% to 25% load operations and mostly less than 10% store operations. It can be noted that Raytrace is the only application that has a considerable amount of stores of 29%, which is even greater than the percentage load operations in Raytrace. Most of these workloads have a high percentage of branch instructions. The basic block sizes vary between 4 instructions to at most 18 instructions, showing that these workloads will be quite sensitive to the branch predictability of a machine configuration.

For the synchronization characteristics, the calls to the system call functions pthread_mutex_lock and pthread_mutex_unlock are recorded using the instruction trace. The number of instructions between the lock and unlock calls is recorded for the size of the critical section. Inside the lock and unlock function calls, the mutex object address to which the exclusive locks and unlocks happen are also recorded. Based on the number of such unique addresses accessed, one can quantify the conflict density in terms of the synchronization events happening across various threads of the workload. All the synchronization metrics are recorded relative to the dynamic number of instructions to be able to replay it in the synthetic to clone these workloads.

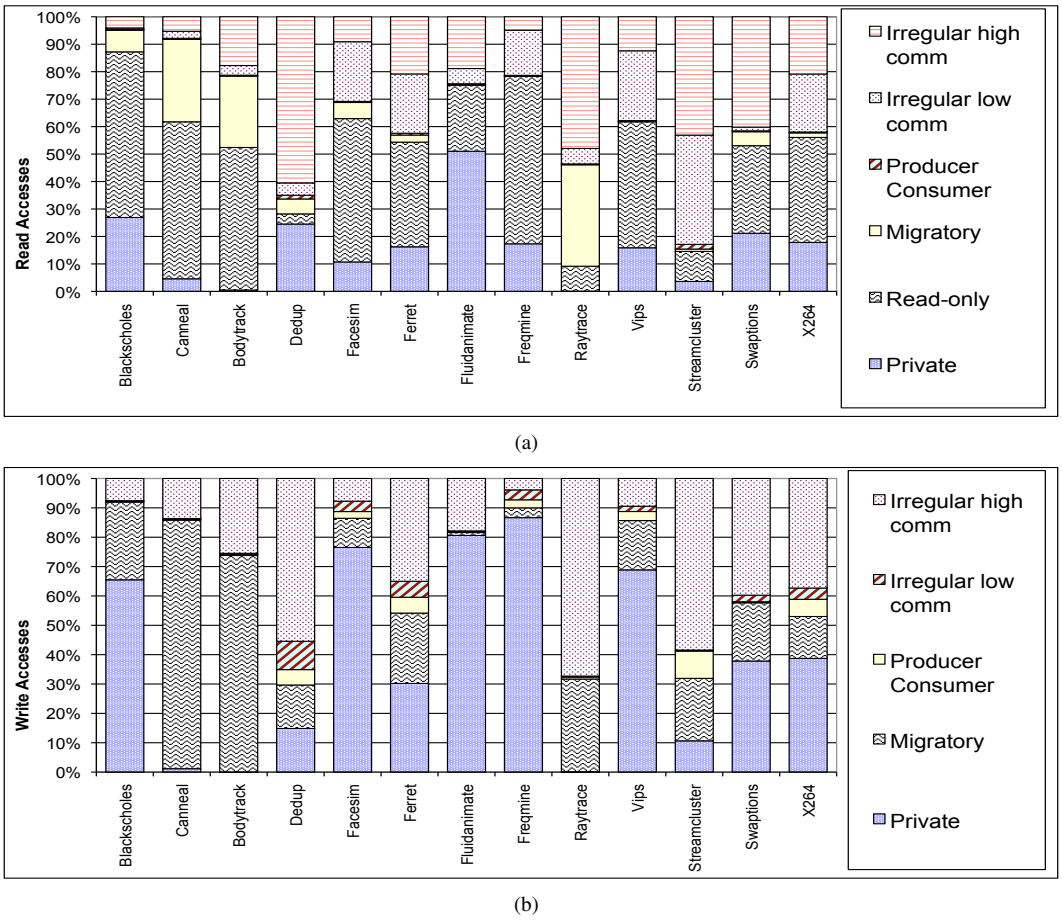


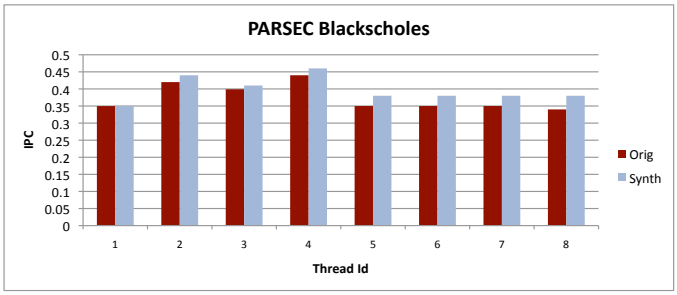
Fig. 8. Temporal distribution of the various memory accesses in the PARSEC workloads into different sharing patterns for (a) Reads (b) Writes

The memory access trace is post processed to record the memory access strides for each of the static load store instructions. The same memory access trace is then post processed and a memory map is built. If an address is not accessed by more than one thread, the address is classified as a private memory location. Further each shared address is further examined to classify them into producer-consumer, migratory and read-only sharing patterns. If an address does not show any conceivable pattern, they are classified to be following an irregular access pattern. Based on the spatial distribution of this data, i.e, the number of addresses that belong to each of the sharing pattern classes, the data footprint of the synthetic will also be distributed. The category irregular is once again broken into two classes, one which has a high communication overhead and the one that has low communication overhead based on the fact whether the data is accessed by more than one processor within a given number of accesses. The Figure 7(b) shows this spatial distribution of the accesses in terms of various sharing patterns. Based on this data, we can see that the private data footprint of applications like Blacksholes, Facesim, Fluidanimate, Freqmine, Vips and X264 are considerably high compared to the shared data footprint. The workloads Canneal, Bodytrack, Raytrace and Streamcluster are the only workloads where the shared data footprint is higher than the private data footprint. In the cases of Canneal and Bodytrack, the read-only shared data content is considerably high compared to other workloads. Bodytrack also has a considerable amount of data that are classified into migratory pattern.

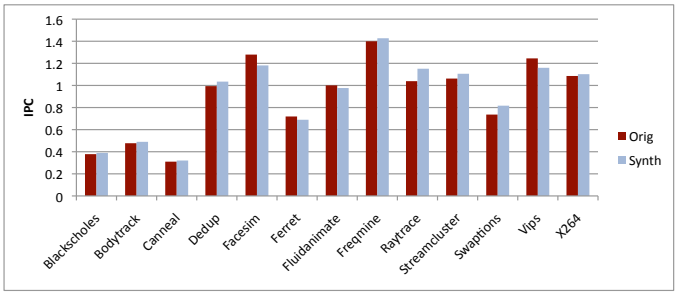
Then, based on the number of accesses to each of these addresses, the proportion of load accesses and store accesses to each of these sharing patterns is determined. This information is much more important than the spatial distribution of the data into different sharing patterns. Many workloads may not have a huge shared data footprint, but can have more shared data accesses than private data accesses. The Figures 8(a) and 8(b) show this temporal distribution of the accesses into various sharing patterns. Good examples of workloads with a low shared data footprint, but with a high amount of accesses to shared data are Facesim, Freqmine and Vips. The memory level parallelism, control flow predictability metrics are recorded using the information provided by the processor and memory models in the GEMS infrastructure.

4 Results and Analysis

The characterized data for the PARSEC applications are fed to the synthetic benchmark generation framework to generate proxies for the different PARSEC applications. Total dynamic instructions in the synthetic vary between three to ten million

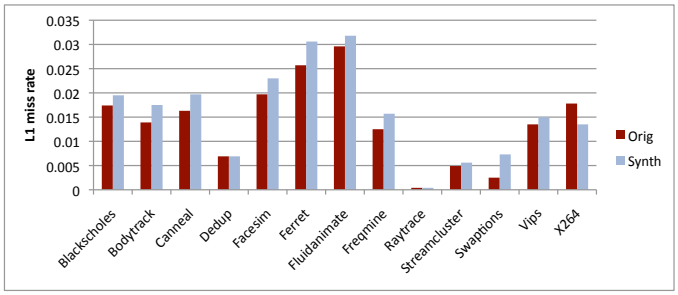


(a)

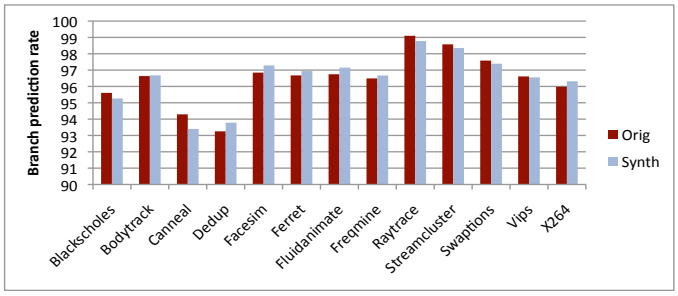


(b)

Fig. 9. (a) Comparison of IPC between original and synthetic for various threads of benchmark Blackscholes in the PARSEC suite (b) Comparison of IPC between original and synthetic for various PARSEC benchmarks



(a)



(b)

Fig. 10. (a)Comparison of L1 missrate between the synthetic proxies and that of the original PARSEC workloads (b) Comparison of branch prediction rate between the synthetic proxies and that of the original PARSEC workloads

for the results provided in this paper. To be able to determine the efficacy of the cloning methodology for multithreaded applications, the next step is to assess the representativeness of the generated proxies to that of their original counterparts. This is accomplished by comparing the performance and power characteristics of the proxies to the original applications.

4.1 Simulation Infrastructure for Evaluation

For all our performance and power estimation simulations, we use the Wisconsin GEMS [30] infrastructure with the Windriver Simics full system simulator. We use the detailed our-of-order processor model Opal with the power model Wattch [31] to model the cores. We use the Ruby tool provided by GEMS to model the memory hierarchy in terms of the caches and the DRAMsim [32] simulator from Maryland to model the DRAM. To model the power consumption of caches, we use the CACTI tool from HP. For modeling the performance of the interconnection network, we use the Garnet simulation model and for modeling the power consumption, we use the Orion [33] power model from Princeton. All our power estimations are done using a 90nm technology. We use the precompiled binaries with gcc-hooks for the SPARC ISA provided by the PARSEC authors at Princeton on Solaris 10 operating system.

4.2 Accuracy in Assessing Performance

A typical 8-core modern system configuration is used for the comparison of various metrics between the synthetic and the original. The machine configuration that is used has an 8GB DRAM, 32KB, 4-way, 1 cycle latency L1 cache, a 4MB, 8-way 8-banked L2 cache, 32 MSHRS, 64 entry reorder buffer with a machine width of 4 instructions per cycle. The branch predictor used is a YAGS branch predictor with a 11 bit addressable Pattern History Table (PHT). The configuration has a 512 entry Branch Target Buffer (BTB), 3 integer ALUs with one integer divide, 2 floating point ALUs with one FP multiply and one FP divide units. The topology that is used to connect the various memory components is a hierarchical switch. The original and the synthetic workloads are run on this configuration and the execution time in terms of number of cycles is recorded. We do not require any warmup for the caches when running the synthetic workload as the caches tend to get warmed up pretty quickly due to the relatively smaller data footprint used in the synthetic. Based on the number of instructions executed, the Instruction-Per-Cycle is computed for each of the threads. Figure 9(a) shows the comparison of IPC between the synthetic and the original for various threads of a randomly chosen PARSEC benchmark Blackscholes. The average error when considering all the threads is 2.9% for Blackscholes. Figure 9(b) shows the comparison of IPC between original and synthetic averaged over all the threads for the various benchmarks in the PARSEC suite. The error in the IPC when averaged over all the 13 benchmarks is 4.87% with a maximum error of 10.8% for the workload Raytrace. It should be noted that Raytrace is unique in terms of the number of writes that it does to memory as previously discussed about the instruction mix of Raytrace.

Other microarchitecture metrics like the miss rates in L1 and the branch prediction rates are also compared between the

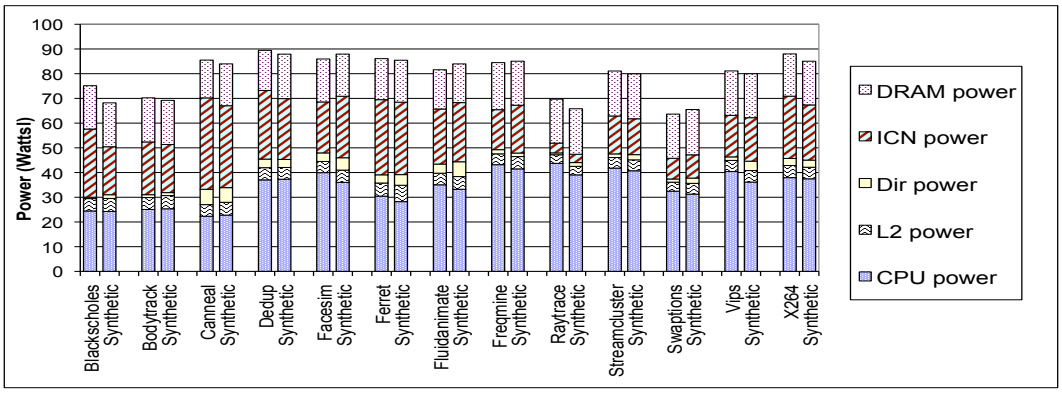


Fig. 11. Power-per-cycle for various PARSEC workloads along with a breakdown of the power consumption in various components on a 8-core system

original and synthetic workloads for various PARSEC applications. The Figure 10(a) shows the comparison of the L1 cache missrate between the original and the synthetic applications. Since the L1 missrates for many of the applications are quite small, the average in the L1 hit rate is computed to assess the representativeness. The average error in the L1 hit rate across all the PARSEC workloads is 0.67% with a maximum of 1.83% for the application Facesim. The Figure 10(b) shows the comparison of the branch prediction rate between the original and the synthetic applications. The average error in the branch prediction rate is 0.52%.

4.3 Accuracy in Assessing Power Consumption

To see how effectively the synthetic benchmarks can be used as proxies for the original PARSEC workloads for power modeling, the power consumption of various workloads is compared to that of their synthetic proxies. Figure 11 shows the comparison of the total system power consumption between the original and synthetic workloads. To show how effectively the synthetic models the execution behavior in each of the system components, the same figure is also annotated with the breakdown of the power consumption in the different system components for the synthetics and the originals. The average error in the total power consumption between the synthetic and the original workloads is 2.73% with a maximum error of 5.5% for the application Raytrace. It should be noted that Raytrace is the application that also has a maximum error in performance and in most of the cases the power consumption of workloads are quite proportional to their performance.

4.4 Accuracy in Assessing Sensitivity to Design Changes

In computer architecture, estimating the performance of a workload on one machine configuration is less important comparing to the ability to estimate the sensitivity of a workload's performance to various design changes. Thus, it is important to evaluate the representativeness of the synthetic workloads to their original counterparts in terms of their sensitivity to design changes. To accomplish this, we use

Parameter	System – A	System – B	System – C
No. of cores	8	8	8
DRAM	16 GB	8 GB	4 GB
L1 cache	64 KB, 4 way, 2 cycles	32 KB, 4 way, 1 cycle	16 KB, 2 way, 1 cycle
L2 cache	8 MB, 16 way, 16 banks	4 MB, 8 way, 8 banks	2 MB, 8 way, 8 banks
L1, L2 MSHRs	48	32	24
ROB	128	64	32
Mach-width	8	4	2
Branch pred.	YAGS, 12 bit PHT	YAGS, 11 bit PHT	YAGS, 10 bit PHT
BTB size	1024	512	256
Int ALUs	4 ALU, 2 Int div	3 ALU, 1 Int div	2 ALU, 1 Int div
Topology	Crossbar	Hierarchical switch	Hierarchical switch
FP ALUs	2 ALU, 2 Mul, 2 div	2 ALU, 1 Mul, 1 div	1 ALU, 1 Mul, 1 div

Fig. 12. Multicore machine configurations used to evaluate the accuracy in assessing the impact of design changes by the synthetic in comparison to original PARSEC workloads

three system configurations as shown in Figure 12 to analyze performance variations for design changes. The three system configurations have varying microarchitecture settings in terms of cache sizes, machine width, branch predictor, topology of the interconnection network etc. To make more design points, the system configuration B was mutated as following to form nine more configurations: 0.5X L1 cache size, 2X DRAM size, 2X L2 cache size, 2X machine width and ROB size, 2X PHT size for branch predictor, more ALUS, ICN crossbar and 0.5X L2 cache size. The performance of the workloads on each of these configurations were recorded for both original and synthetic workloads using the metric IPC. The correlation between the trends followed by original and the synthetic is determined by finding the correlation coefficient.

The Figure 13 shows the correlation coefficient between the trends followed by the synthetic and the original workloads for the various design changes. The higher the correlation coefficient, the better is the correlation between the trends. The average of the correlation coefficient for all the workloads in the PARSEC suite is 0.92. The Figures 14(b) and 14(a) show the comparison of sensitivity to design changes using various multicore machine configurations by mutating system configuration B for the randomly chosen workloads Raytrace and Streamcluster in PARSEC suite respectively. This brings out the utility value of the synthetics to be used as proxies for the PARSEC workloads for the most invasive design space

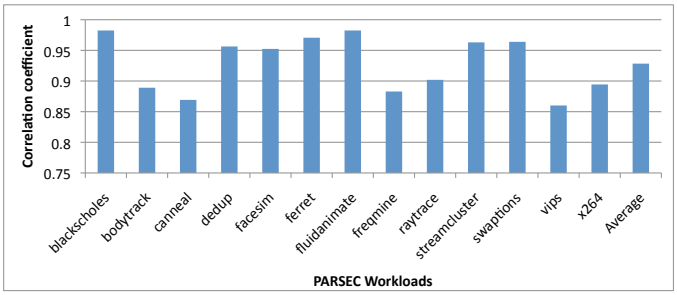
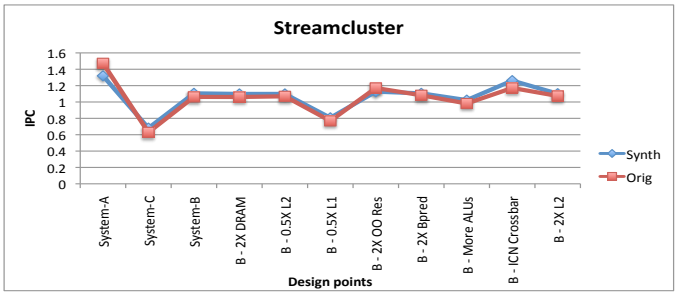
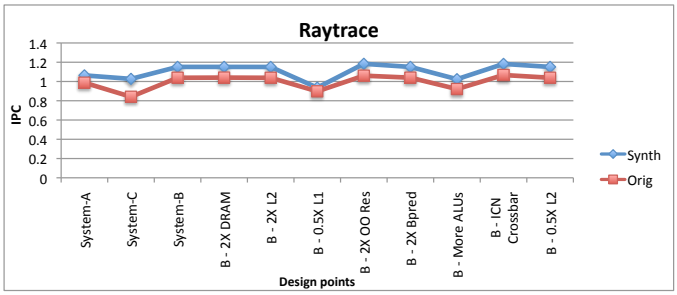


Fig. 13. Correlation coefficients for the sensitivity to design changes between the synthetic and the original using various multicore machine configurations for the workloads in the PARSEC suite



(a)



(b)

Fig. 14. Comparison of sensitivity to design changes using various multicore machine configurations for the workloads (a) Streamcluster and (b) Raytrace in PARSEC suite

exploration studies.

4.5 Speedup Achieved in Using the Synthetics

The most important advantage of using the synthetic proxies over the long running original PARSEC applications is the speedup achieved in simulations when using these miniaturized proxies. The table in Figure 15 shows the speedup achieved in terms of reduction in the number of instructions when using the synthetic proxies over the original PARSEC applications. The synthetic benchmarks generally have three to eight million instructions, when the original workloads have a few thousand million instructions. The speedup achieved is at least four orders of magnitude to a maximum of about five orders of magnitude for the application Facesim.

Benchmarks	No. of Instructions (Millions)		Speedup
	Synthetic	Original	
Blackscholes	3.80	13028	3429
Bodytrack	5.96	56918	9552
Canneal	8.29	10484	1264
Dedup	4.41	8428	1912
Facesim	7.45	1151026	154450
Ferret	4.42	58398	13227
Fluidanimate	7.81	72669	9300
Freqmine	6.93	5588	807
Raytrace	4.02	223560	55585
Streamcluster	4.51	52527	11641
Swaptions	7.01	11020	1572
Vips	7.50	37135	4952
X264	3.75	13001	3465

Fig. 15. Speedup achieved by using the synthetic proxies over the full run of the PARSEC workloads on a 8-core system configuration

4.6 Proxies for Proprietary Applications

As described before, one of the applications of our cloning methodology is to disseminate proprietary applications to processor architects for better performance analysis of target workloads. The most important feature about our synthetic benchmark generation is that they cannot be reverse engineered to find any information about the original applications. There has been a lot of research in code obfuscation techniques [34] [35] [36], where one tries to obfuscate the code to hide the intellectual property in applications when distributing software binaries to make it harder to reverse engineer. Though these techniques have been researched a lot, most of them suffer from increasing the execution time of the program or code size in some way. In our synthetic benchmark generation case, it is to be noted that only the performance characteristics of the workloads are distilled into a synthetic benchmark that does not have any functional meaning, making any kind of reverse engineering quite meaningless. We also obfuscate the organization of the data in the original applications, by converting all the data structures into a single one dimensional array. This removes the information like the presence of a class (in C++ or Java) and the presence of a structure in C. This final piece of code is independent of most of the higher level constructs of loops, function calls and other possible organization in the code. Thus, the proposed synthetic benchmark cloning is quite robust to disseminate proprietary applications without the need to worry about being reverse engineered for information about the original applications.

5 Limitations

The synthetic proxies generated using our methodology are intended to be used by designers, who want to explore the design space in early design stage of a processor. After narrowing down the design space, we recommend using other methods to more accurately estimate the absolute performance for a given design. The synthetic proxies of standard benchmarks should not be used as a sole method to publish final performance numbers. When compiling the synthetic proxies, one should bear in mind that they are not meant to be optimized by the compiler as the characteristics incorporated into them

using embedded assembly should not be altered to be able to reproduce the behavior of the original workload. Thus, the synthetic proxies should not be used for any performance study aiming to evaluate the efficacy of a compiler optimization [37] [38]. And, another limitation of these multithreaded synthetic proxies is that they are generated for a specific number of cores and they need to be regenerated when the number of cores are changed in the design. So, these multithreaded proxies cannot be used to quantify performance differences for studies involving changing the number of cores. When the phase change behavior in a program is critical to a study, these multithreaded synthetics should not be used, because the proposed synthetic benchmark methodology is built upon statistical simulation, where the characteristics of a workload are averaged over the period of execution and this averaged behavior is replayed statistically in the most representative manner. But, please note that it is still possible to profile the various phases (like simulation points) of a workload separately and generate individual synthetic proxies for each of these phases. With minmal effort, the code generated for each of these individual synthetic proxies can all be tied together (as function calls) for replaying phase changes for single threaded workloads.

6 Summary

A characterization of the PARSEC workloads have been provided and miniaturized proxies for these workloads have been generated and provided aiming at solving the problems related to prohibitive runtime and unavailability of proprietary target applications in processor design. These proxies have been validated by assessing their performance in comparison to the original applications on a 8-core typical modern system configuration. The average error in the IPC for these workloads is 4.87% and maximum error is 10.8% for Raytrace in comparison to the original workloads. Similarly, the average errors in the L1 cache hitrates and branch prediction rates are 0.67% and 0.53% respectively. It is also shown that the generated synthetic proxies also have very similar power consumption to that of the original workloads, opening the doors for using these synthetic clones for power modeling. The average error in the power-per-cycle metric is 2.73% with a maximum of 5.5% when compared to original workloads. To further show how faithfully the synthetic benchmark follows the execution behavior of the original workloads in various system components, the breakdown of the power consumption of the synthetic is compared with that of the original workloads. The representativeness of the synthetic proxies to that of the original workloads in terms of their sensitivity to design changes is also shown to be quite good by finding the correlation coefficient as 0.92 between the trends followed by the synthetic and the original for design changes. Finally, the speedup achieved by using the synthetic proxies over the original workloads is shown to be around 4 orders of magnitude and up to 6 orders of magnitude for some specific workloads.

Acknowledgments

This work has been supported and partially funded by SRC under Task ID 1797.001, National Science Foundation under grant numbers 0702694, 0751112, 0750847, 0750851, 0750852, 0750860, 0750868, 0750884, 1117895 and 0751091, Lockheed Martin, Sun Microsystems, IBM and AMD. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or other sponsors.

References

[1] K. Ganesan, D. Panwar, and L. K. John, "Generation,Validation and Analysis of SPEC CPU2006 Simulation Points Based on Branch, Memory, and TLB Characteristics," *SPEC Benchmark Workshop 2009, Austin, TX, Lecture Notes in Computer Science 5419 Springer pages 121-137, January 2009*.

[2] G. Hamerly, E. Perelman, and B. Calder, "How to Use SimPoint to Pick Simulation Points," *ACM SIGMETRICS Performance Evaluation Review*, March 2004.

[3] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," *Proceedings of the International Symposium on Computer Architecture, (ISCA 2003), p. 84 - 95*.

[4] M. V. Biesbrouck, T. Sherwood, and B. Calder, "A co-phase matrix to guide simultaneous multithreading simulation," *In Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS04)*, March 2004.

[5] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite," *The 34th International Symposium on Computer Architecture (ISCA 2007)*, June 2007.

[6] K. Ganesan, J. Jo, and L. K. John, "Synthesizing Memory-Level Parallelism Aware Miniature Clones for SPEC CPU2006 and Implant-Bench Workloads," *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2010.

[7] K. Ganesan, J. Jo, W. L. Bircher, D. Kaseridis, Z. Yu, and L. K. John., "System-level Max Power (SYMPO) - A systematic approach for escalating system-level power consumption using synthetic benchmarks," *In the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), Vienna, Austria*, September 2010.

[8] K. Ganesan and L. K. John, "MAximum Multicore POwer (MAMPO) - An Automatic Multithreaded Synthetic Power Virus Generation Framework for Multicore Systems," *In the SuperComputing Conference (SC 2011), Seattle, WA*, November 2011.

[9] L. John, J. Jo, and K. Ganesan, "Workload Synthesis for a Communications SoC," *In Workshop on SoC Architecture, Accelerators and Workloads, held in conjunction with HPCA-17, San Antonio, Texas*, February 2011.

[10] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "SimPoint 3.0: Faster and More Flexible Program Analysis," *Workshop on Modeling, Benchmarking and Simulation*, June 2005.

[11] M. Oskin, F. T. Chong, and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Design," *In Proceedings of the International Symposium on Computer Architecture (ISCA 2000)*, 2000.

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

[13] L. Eeckhout, R. H. B. Jr., B. Stougie, K. D. Bosschere, and L. K. John, "Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies," *Proceedings. 31st Annual International Symposium on Computer Architecture, (ISCA 2004)*, 2004.

[14] W. S. Wong and R. J. T. Morris, "Benchmark Synthesis Using the LRU Cache Hit Function," *IEEE Transactions on Computers*, 1988.

[15] G. Balakrishnan and Y. Solihin, "WEST: Cloning data cache behavior using Stochastic Traces," *IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*, February 2012.

[16] J. Robert H. Bell, R. R. Bhatia, L. K. John, J. Stuecheli, J. Griswell, P. Tu, L. Capps, A. Blanchard, and R. Thai., "Automatic Testcase Synthesis and Performance Model Validation for High Performance PowerPC Processors," *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2006)*, March 2006.

[17] A. Joshi, L. Eeckhout, R. H. B. Jr., and L. K. John, "Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks," *International Symposium on Workload Characterization*, October 2006.

[18] K. Ganesan, "Automatic Generation of Synthetic Workloads for Multicore Systems," *Department of Electrical and Computer Engineering, The University of Texas at Austin,*, December 2011.

[19] Z. Jin and A. C. Cheng, "ImplantBench: Characterizing and Projecting Representative Benchmarks for Emerging Bio-Implantable Computing," *IEEE Micro (IEEE Micro), 28(4):71-91*, July/August 2008.

[20] A. Joshi, L. Eeckhout, L. K. John, and C. Isen, "Automated microprocessor stressmark generation," *The 14th International Symposium on High Performance Computer Architecture (HPCA)*, February 2008.

[21] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[22] A. Joshi, L. Eeckhout, J. Robert H. Bell, and L. K. John, "Distilling the essence of proprietary workloads into miniature benchmarks," *ACM Transactions on Architecture and Code Optimization (TACO 2008)*, August 2008.

[23] R. H. Bell and L. K. John, "Improved Automatic Test Case Synthesis For Performance Model Validation," *Proceedings of the International Conference on Supercomputing 111-120*, 2005.

[24] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication Characterization of Splash-2 and Parsec," *IEEE International Symposium on Workload Characterization*, October 2009.

[25] M. C. H. Hemayet Hossain, Sandhya Dwarkadas, "Improving support for Locality and fine-grain sharing in chip multiprocessors," *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, October 2008.

[26] L. Cheng, J. B. Carter, and D. Dai, "An Adaptive Cache Coherence Protocol Optimized for Producer-Consumer Sharing," *IEEE 13th International Symposium on High Performance Computer Architecture, 2007. HPCA 2007*, February 2007.

[27] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak, "Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors," *Proceedings of the IEEE/ACM Supercomputing Conference*, 1995.

[28] G. Viswanathan and J. R. Larus, "Compiler-directed Shared-Memory Communication for Iterative Parallel Applications," *Proceedings of the ACM/IEEE Conference on Supercomputing*, 1996.

[29] H. M, S. P, and F. M, "Branch transition rate: a new metric for improved branchclassification analysis," *Sixth International Symposium on High-Performance Computer Architecture (HPCA 2000), Volume , Issue , 2000 Page(s):241 - 25*, January 2000.

[30] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, , and D. A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News (CAN),*, September 2005.

[31] M. Martonosi, V. Tiwari, and D. Brooks, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *isca, pp.83, 27th Annual International Symposium on Computer Architecture (ISCA 2000)*.

[32] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "DRAMsim: A memory-system simulator," *Computer Arch. News, vol. 33, no. 4, pp. 100-107*, Sep 2005.

[33] H. Wang, X. Zhu, L.-S. Peh, and S. Malik, "Orion: A Power-Performance Simulator for Interconnection Networks," *In Proceedings of MICRO 35, Istanbul, Turkey*, November 2002.

[34] E. Jokipii, "Jobe - The Java obfuscator - http://www.primenet.com/~ej/index.html," 1996.

[35] J. J. Marciniak, "Encyclopedia of Software Engineering," *chapter Reverse Engineering, pp 1077-1084. John Wiley & Sons, Inc, 1994. ISBN 0-471-54004-8.*

[36] A. Herzberg and S. S. Pinter, "Public protection of software," *ACM Transactions on Computer Systems, vol. 5, no. 4, pp 371-393, November 1987.*

[37] K. Ganesan, L. K. John, J. Sexton, and V. Salapura, "A Performance Counter Based Workload Characterization on BlueGene/P," *In 37th International Conference on Parallel Processing (ICPP), Portland, Oregon*, September 2008.

[38] V. Salapura, K. Ganesan, A. Gara, M. G. J. C. Sexton, and R. E. Walkup, "Next-Generation Performance Counters: Monitoring Over Thousand Concurrent Events," *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium, pages 139-146*, April 2008.

Karthik Ganesan received his PhD in Computer Engineering from the University of Texas at Austin in 2011. He received his Master of Science in Computer Engineering from the University of Texas at Austin in 2008 and Bachelor of Engineering in Computer Science and Engineering from Anna University, India in 2006. Karthik Ganesan is currently a senior member at Oracle America Inc, and his research focuses on the performance evaluation of Oracle server systems. Karthik has internship experiences in the BlueGene design team at IBM T. J. Watson labs working with the then world's fastest supercomputer. He also has internship experiences in the Performance Inspector tool development team at IBM Austin and Cortex processor design team at ARM Inc. During his PhD, he was working as a research assistant in the Laboratory for Computer Architecture directed by Prof. Lizy K John. His research interests include multicore system architectures, performance evaluation, benchmarking and Java performance. During his undergraduate study, he was also working as a part time research trainee at Waran Research Foundation, India. He has authored several papers published in top tier international conferences and journals. He also actively serves as a technical reviewer for many such conferences and journals. Karthik Ganesan is a member of the IEEE and the Association for Computing Machinery (ACM).

Lizy Kurian John is the B. N. Gafford Professor in Electrical Engineering in the ECE Department at UT Austin. She received her Ph. D in Computer Engineering from the Pennsylvania State University in 1993. Her research interests include high performance processor and memory architectures, low power design, reconfigurable architectures, rapid prototyping, Field Programmable Gate Arrays, workload characterization, etc. She is recipient of NSF CAREER award, UT Austin Engineering Foundation Faculty Award (2001), Halliburton, Brown and Root Engineering Foundation Young Faculty Award (1999), University of Texas Alumni Association (Texas Exes) Teaching Award (2004), The Pennsylvania State University Outstanding Engineering Alumnus (2011) etc. She has coauthored a book on Digital Systems Design using VHDL (Thomson Publishers, 2007) and has edited 4 books including a book on Computer Performance Evaluation and Benchmarking. She holds 7 US patents and is a Fellow of IEEE.