Synthesizing Memory-Level Parallelism Aware Miniature Clones for SPEC CPU2006 and ImplantBench Workloads

Karthik Ganesan, Jungho Jo and Lizy K John
Department of Electrical and Computer Engineering, University of Texas at Austin,
{*karthik, jungho.jo*}*@mail.utexas.edu, ljohn@ece.utexas.edu*

***Abstract*—We generate and provide miniature synthetic benchmark clones for modern workloads to solve two pre-silicon design challenges, namely: 1) huge simulation time (weeks to months) when using complete runs of modern workloads like SPEC CPU2006 having trillions of instructions on pre-silicon design models 2) unavailability of access to their specific target applications for computer architects, as some of them are proprietary in nature and vendors hesitate to share them. We first provide a detailed characterization of the SPEC CPU2006 and the ImplantBench suites based on microarchitecture-independent metrics. Our metrics include the Memory Level Parallelism (MLP) of these workloads to estimate the burstiness of accesses to the main memory. Secondly, our proposed framework, that uses this characterized information (including MLP) to generate synthetic clones is explained and evaluated. We provide the synthetic clones generated for CPU2006 workloads for download and use. The efficacy of the synthetic clones for CPU2006 and ImplantBench is demonstrated by comparing their performance and power characteristics with their original counterparts. We show that the synthetic clones 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. 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). For ImplantBench, we have an average error of 2.9% in assessing the IPC and the correlation coefficient for assessing design changes is 0.94 (0.97 for power-per-cycle).**

I. Introduction

The SPEC CPU2006 [1] suite released in August 2006 contains many programs, each with thousands of billions of dynamic instructions taking weeks to months of time on cycle accurate simulators like the SimpleScalar [2] as detailed by Ganesan et al [3]. Pre-silicon design models are many orders slower than cycle accurate simulators and it is almost impossible to use full runs of such modern workloads/benchmarks for design space exploration. The advent of the multicore processors and heterogeneity in the cores has made the design space exploration even more challenging, resulting in a prohibitively large amount of simulation time. This has driven architects to use samples/traces of important parts of the target applications instead of complete runs. It is to be noted that even after 3 years after the release of the SPEC CPU2006 suite, we do not see many simulation based papers using these more representative modern workloads and rather architects tend to use the older version CPU2000 due to the availability of miniaturized samples/traces.

We generate and provide miniaturized synthetic benchmark clones using an improved methodology for these huge SPEC CPU2006 workloads and also clones for the futuristic embedded workloads in the ImplantBench suite. During the pre-silicon design stage of a processor, having miniaturized synthetic clones (not more than 1 million dynamic instructions) for large target applications enables an architect to use them with slow low level simulation models (e.g., RTL models in VHDL/Verilog) and helps in making confident decisions in designing processors tailored for the targeted applications. Another important application of synthetic benchmark clones is that they can be distributed to architects and designers even if the original applications are intellectual property that are not publicly available. The synthetic benchmark clones are carefully generated such that they do not reveal the functionality of the original application, but still capture the essence of the performance and power characteristics of the application. Today's ever increasing number of applications and a need to design processors tailored to a particular class of applications along with a faster 'time-to-market' necessitates the need for a framework to automate the process of generating synthetic benchmark clones for the target workloads. Such a framework will enable architects to be 'up-to-date' with their applications, as they keep evolving and also have proxies for futuristic applications generated. These synthetic clones are highly space efficient and easy to use compared to other simulation time reduction techniques (elaborated in Section-2).

Previous efforts towards synthesizing workloads [4] [5] utilize the memory access, control flow and the instruction level parallelism information of the original workload, but do not characterize or use the miss pattern information of the last level cache, viz., Memory Level Parallelism (MLP) information. As a result, the synthetics generated using these previous approaches always have misses in the last level cache happening at a constant frequency without much burstiness. When the original workload has high MLP (bursty misses), the generated synthetic results in having an entirely different execution behavior compared to the original workload as shown

978-1-4244-6022-9/10/$26.00 ©2010 IEEE

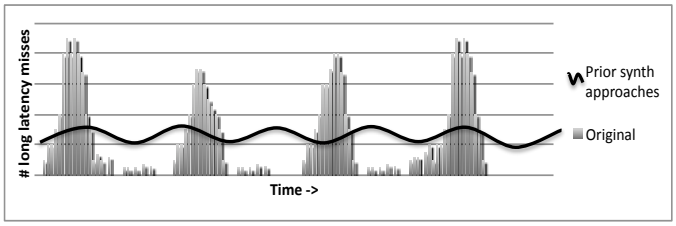


Fig. 1. Comparison of the MLP behavior of synthetics generated by previous approaches to that of a typical workload

in Figure 1. Our synthetic benchmark generation methodology overcomes this important shortcoming by incorporating the MLP information to model the burstiness of accesses to the main memory and significantly improves the representativeness of the clone. We show that the synthetic benchmarks generated using our methodology have 12% more accuracy in their representativeness to the original application in terms of IPC compared to the previous approaches [4] and [5]. The significant contributions of this paper are,

- We provide an improved methodology for synthetic benchmark generation incorporating the MLP information, using multiple loops and modeling load-load dependencies to generate miniaturized synthetic clones for any target workload and more importantly provide the clones [6] for SPEC CPU2006 workloads for the architecture community to download and to use.
- We provide a detailed microarchitecture-independent characterization of the CPU2006 and ImplantBench workloads.
- We also provide the accuracy in using the provided synthetic clones as proxies to evaluate the performance and power consumption of the original CPU2006 and the ImplantBench workloads for a typical modern microarchitecture.
- We also validate the sensitivity of these synthetic benchmarks to microarchitecture design changes with the sensitivity of their corresponding original counterparts.

The rest of the paper is organized as follows. In Section 2, we briefly describe the related work, elaborating on the previous work in synthetic benchmark generation and the motivation behind an MLP aware synthetic generation. Section 3 provides the summary of the information captured in the process of characterizing the workloads under study and explains our framework to generate the synthetic clones. Section 4 elaborates on the experimental results regarding the representativeness of the generated synthetic clones to that of the original benchmarks. We summarize and provide future directions in Section 5.

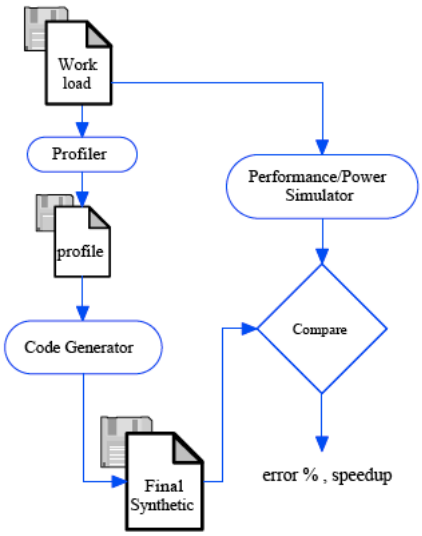
II. Related Work and Motivation

To reduce simulation time, sampling techniques like simulation points [7] and SMARTS [8] are well known and widely used. But, the problem with such sampling techniques is that huge trace files for the particular dynamic execution interval have to be stored or requires the simulator to have the capability to fast-forward until it reaches the particular interval of execution that is of interest to the user. But rather, the synthetic benchmarks that we provide are space efficient in terms of storage and do not require any special capability in a simulator. The problem with other techniques like benchmark subsetting [9] is that the results are still whole programs and are too big to be directly used with pre-silicon design models.

Oskin et al. [10] and Nussbaum et al. [11] introduced the idea of statistical simulation to guide the process of design space exploration. Eeckhout et al [12] proposed the use of Statistical Flow Graphs (SFG) in characterizing the control flow behavior of a program in terms of the execution frequency of basic blocks annotated with their mutual transition probabilities. Wong et al. introduced the idea of synthesizing benchmarks [13] [14] [15] based on the workload profiles. Bell and John [5] and Joshi et al. [4] synthesized benchmark clones for the workloads in the SPEC CPU2000 suite by using a technique in which one loop is populated with embedded assembly instructions based on the instruction mix, control flow behavior, the memory behavior and the branch behavior of the original workload. This generated synthetic loop was iterated until the performance characteristics became stable.

The memory model used by the previous approaches [4] [5] consists of a set of static loads/stores that access a series of memory locations in a stride based access pattern. Even though the loads within this single loop are populated in such a way that they match the miss rates of the original application, they may not necessarily match the performance of the original application precisely. We classify loads into two categories. The loads that miss in the last level of the on-chip cache and result in an off-chip memory access are called 'long-latency' loads and the other set of loads that hit in the caches. Since these previous synthetic benchmark generation approaches do not model the burstiness of these long-latency loads, the long-latency loads are distributed randomly throughout the synthetic loop. These long-latency loads keep missing in a constant frequency as this loop is being iterated without much overlap in their execution. But the original workloads with the same miss rates may not necessarily have such a behavior. As already shown in Figure 1, the typical memory access behavior of the synthetics generated by the previous techniques can be entirely different compared to the case of many of the original workloads. 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. In the original, though the pipeline may be clogged in this first interval due to the long-latency miss, the instructions may flow freely through the pipeline in the second. Rather, in the synthetic generated by previous approaches, there is a constant clog in the pipeline throughout the execution resulting in an entirely different execution behavior. In Section 3, we characterize the burstiness of misses in the target workloads and show real cases with the behavior (high MLP) as shown in Figure 1.

Since a long-latency load incurs hundreds of cycles due to the off-chip memory access, the performance of a workload



(a)

No.	Metric	Category
1	Dynamic execution freq. of basic blocks	Control flow predictability
2	Transition probabilities in the SFG	
3	Average basic block size	
4	Branch taken rate for each branch	
5	Instruction pattern in a basic block	
6	Branch transition rate for each branch present in the workload	
7	% Integer multiplication	Instruction mix
8	% Integer division	
9	% FP multiplication	
10	%FP division	
11	% Loads	
12	% Stores	
13	% Branches	
14	Dependency distance distribution per instruction type	Instruction level parallelism
15	Stride value per static load/store	Data locality
16	Data Footprint of the workload	
17	**Mean and standard deviation of the MLP**	**Memory level parallelism**
18	**Mean number of consecutive dynamic inst. when no outstanding long-latency loads**	

(b)

Fig. 2. (a) Overall methodology (b) Metrics profiled to characterize the workloads

varies significantly based on the amount of overlap present in the execution of these long-latency load instructions. The average number of such long-latency loads outstanding when there is at least one long-latency load outstanding is called the Memory Level Parallelism (MLP) present in a workload. Both of the cited previous approaches only characterize and model the Instruction-Level-parallelism in the workloads and fail to characterize and to model the Memory Level Parallelism (MLP) in the workloads. Eyerman and Eeckhout [16] show the impact of MLP on the overall performance of a workload. They show that there can be performance improvements ranging from 10% to 95% for various SPEC CPU2000 workloads if we harness the amount of MLP in the applications efficiently. This brings out the importance of characterizing the MLP in workloads. We characterize and model this MLP information in our synthetic generation framework. For some workloads, we also require more than one loop to mimic the MLP behavior of the original workloads, upon which we elaborate in Section 3.

Further in this Section, we provide some background on the ImplantBench suite. The ImplantBench suite proposed by Jin et al. [17] is a collection of futuristic applications that will be used in bio-implantable devices. Bio-implantable devices are planted into human body to collect, process and communicate realtime data to aid human beings in recovering from various types of defects. A few examples are retina implants, functional electrical stimulation implants and deep brain stimulation implants. ImplantBench is a collection of applications falling into the categories: security, reliability, bioinformatics, genomics, physiology and heart activity. Security algorithms are used in these devices for a safe and secure transfer of data from these implanted devices to the outside world. Reliability algorithms take care of the integrity of the data transferred to and from the implanted devices due to using wireless techniques. Bioinformatics applications are the ones that extract and analyze genomic information. At times a part of a genomic application may be added into the implanted device for some real time uses. Physiology includes the job of collecting and analyzing physiological signals like Electrocardiography (ECG) and Electroencephalography (EEG). Heart activity applications diagnose heart problems by analyzing the heart activity. Jin et al. [17] provide a detailed characterization of these applications, but most of their characterization is based on microarchitecture dependent metrics, whereas our characterization is mostly independent of the microarchitecture.

III. Cloning Framework

Figure 2(a) shows the cloning methodology that is used in this paper. As the first step, the target application is profiled to collect a wide range of characteristics. Then, this information is fed to the code generator to generate the synthetic. This final synthetic is compared with the original and the accuracies are reported.

A. Benchmark Characterization

To capture the various profile information of the workloads, we use modified versions of the different simulators in the SimpleScalar [2] simulation framework. Figure 2(b) delineates the different metrics that are recorded for each of the workloads. Further in this Section, we explain each of these metrics and in tandem, provide the corresponding information captured for each of the metrics for the target workloads.

To capture the control flow behavior of a workload, the locality in the underlying static code being executed needs to be captured. A Statistical Flow Graph (SFG) [12] of the workload is constructed for capturing the control flow behavior of the workload. A SFG consists of nodes that are the basic blocks in the program and the edges represent the mutual transition probabilities between the basic blocks. We also record the average and the standard deviation of the size of the basic

Benchmark	No. of B.Blks	# B.Blks for 90% of Prog Exec.	Branch Transition Rate	Average B.Blk Size	Average Number of Succ. B.Blks
CINT 2006					
400.perlbench	1620	169	0.07	5.99	1.93
401.bzip2	93	15	0.06	7.91	1.94
445.gobmk	617	129	0.15	6.97	2.42
456.hmmer	142	12	0.10	8.12	1.49
458.sjeng	281	70	0.17	5.97	2.54
462.libquantum	29	3	0.03	4.59	1.41
464.h264ref	1074	55	0.09	15.13	1.69
471.omnetpp	443	112	0.08	5.25	2.06
473.astar	96	16	0.19	8.77	1.43
483.xalancbmk	62	11	0.00	3.54	2.38
429.mcf	179	96	0.01	10.58	1.21
403.gcc	698	363	0.08	7.14	1.76
CFP 2006					
410.bwaves	32	10	0.26	32.61	1.50
433.milc	42	18	0.44	15.32	1.48
434.zeusmp	48	13	0.02	43.29	1.67
435.gromacs	6	3	0.07	518.06	1.33
436.cactusADM	22	7	0.18	247.16	1.27
437.leslie3d	533	13	0.02	20.43	1.74
444.namd	70	8	0.03	18.95	1.61
450.soplex	358	14	0.03	7.79	1.45
459.GemsFDTD	119	1	0.01	48.81	1.40
482.sphinx3	544	20	0.07	12.76	1.70

(a) SPEC CPU2006 workloads

Benchmark	No. of B.Blks	# B.Blks for 90% of Prog. Exec.	Branch Transiti on Rate	Average B.Blk Size	Average # of Succ. B.Blks
AI_Adaline	342	6	0.05	39.22	1.63
AI_BPN	410	13	0.06	54.27	1.80
AI_GA	503	160	0.23	7.7	2.26
Bioinformatics_ELO	194	15	0.05	9.42	1.72
Bioinformatics_LMGC	498	111	0.25	7.55	1.91
Genomics_HMM	463	119	0.28	8.03	1.80
Genomics_NJ	196	18	0.17	9.41	2.03
HeartActivity_pNNx	479	97	0.17	6.41	1.68
Physiology_AFVP	679	105	0.26	7.07	1.77
Physiology_ECGSYN	647	142	0.18	8.99	1.65
Reliability_alder32	143	101	0.43	6.81	1.88
Reliability_crc	172	89	0.36	6.1	2.20
Reliability_luhn	155	90	0.40	6.46	2.10
Reliability_reed_solm	255	4	0.02	4.71	1.93
Security_haval	338	82	0.25	8.1	1.94
Security_KHAZAD	198	17	0.14	121.65	1.97
Security_sha2	313	7	0.06	37.32	1.76

(b) ImplantBench workloads

Fig. 3. Captured SFG information and branch transition rate

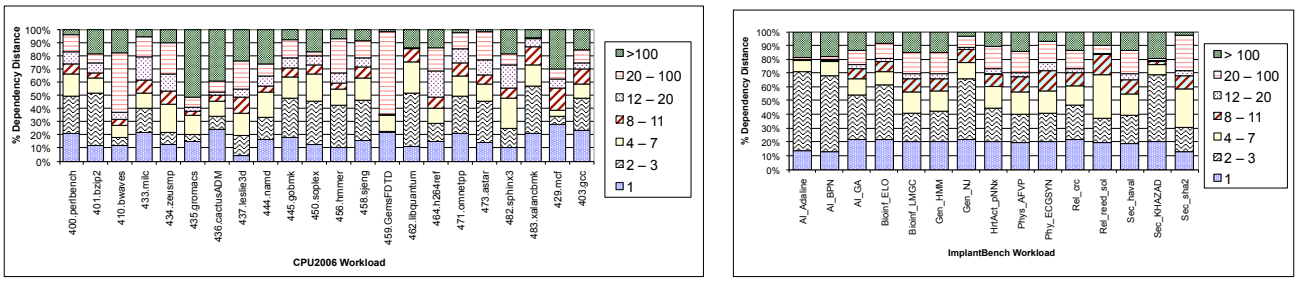
block along with the instruction pattern in the basic block in terms of the instruction type. The instruction mix of the original workload is a significant microarchitecture-independent metric that captures the frequencies of various instruction types, namely: integer ALU operation, integer multiply, integer divide, floating point ALU operation, floating point multiply, floating point divide, load, store and branch.

For our experiments, we use the alpha binaries generated on an alpha machine running the Tru64 UNIX operating system using gcc 4.2 with an optimization level of -O2. A few of the SPEC CPU2006 workloads could not be compiled on the alpha architecture and we show the results for a set of 22 SPEC CPU2006 workloads. The tables in Figures 3(a) and 3(b) show the SFG information captured for the most representative 100 million instruction simulation point of the workloads in the CPU2006 benchmark suite and that of the ImplantBench suite respectively. We can see that the number of basic blocks that account for 90% of execution are only 6% of the total number of basic blocks in that interval of execution showing the amount of redundancy. The tables also show the average basic block size calculated based on both the number of instructions in each of the basic blocks and their dynamic execution frequency. As seen in 3(a), the floating point benchmarks of CPU2006 tend to have bigger basic block sizes compared to their integer counterparts in the same suite. The average number of successor basic blocks is another measure of the control flow complexity of the program. Programs that have complicated switch statements result in more successors and predicting the control flow becomes complicated. Also, when a function is called at multiple sites and each time it returns to different locations, it results in more successors.

The branch predictability of the benchmark can be captured independent of the microarchitecture by using the branch transition rate [18]. 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. The branch transition rates for the CPU2006 workloads as given in Figure 3(a) average around 0.11 with few benchmarks like 433.milc, 410.bwaves having a transition rate above 0.25. Similarly, the branch transition rates of the ImplantBench suite are shown in Figure 3(b). It can be noted that the a few reliability applications have branches with high transition rates.

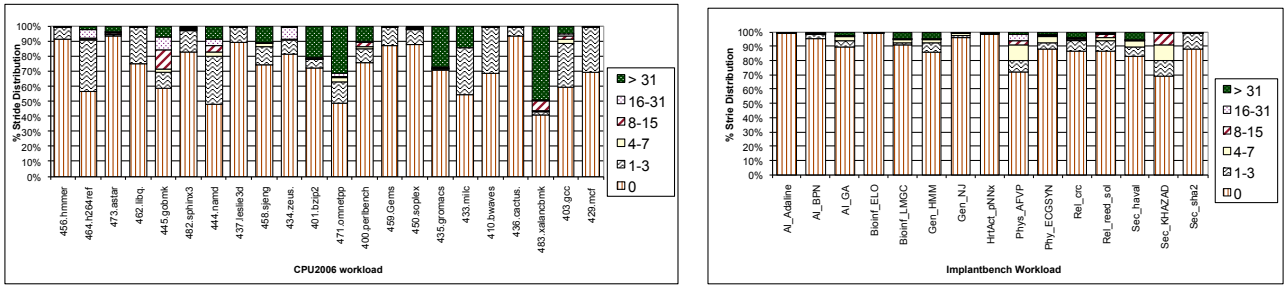
To capture the Instruction Level Parallelism (ILP) in the workload, we capture the dependency distance or the register reuse distance of the workload as a distribution. This information is captured for each type of instruction. This corresponds to the number of instructions between the production and the consumption of a data value at the register level. The proportion of instructions that have an immediate operand is also recorded along with this distribution. This distribution is binned at a granularity of 1, 2, ... 20, 20-100 and greater than 100. The Figures 4(a) and 4(b) show a histogram of the dependency distances for the workloads in CPU2006 and the ImplantBench respectively. It can be observed that a few benchmarks like 436.cactusADM and 435.gromacs, which have very large basic block sizes (518 and 247 respectively), tend to have larger dependency distances. 435.gromacs and 436.cactusADM have respectively 50% and 40% of their dependencies that can be potentially resolved before 100 instructions. Still, it is to be noted that 436.cactusADM has more than 20% of dependencies just before one instruction. Such benchmarks with high instruction level parallelism will be sensitive to the out-of-order resources available in a processor and modeling their dependency distance distribution plays a key role in mimicking the behavior of the original workload. In general, it can be observed that the ImplantBench and SPEC CINT2006 workloads that have smaller basic block sizes have shorter dependency distances compared to the SPEC CFP2006 workloads that have larger basic block sizes. For the ImplantBench and the CINT2006 workloads, more than 50% of their dependency distances are within 2-3 instructions.

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



(a) SPEC CPU2006 (b) ImplantBench

Fig. 4. Dependency distance distribution



(a) SPEC CPU2006 (b) ImplantBench

Fig. 5. Stride based memory access behavior in the target workloads

of the memory hierarchy. Though locality is a global metric characterizing the memory behavior of the whole program, previous studies [4] have resorted to characterizing the access behavior at per static load/store basis in terms of strides (differences between two consecutive effective addresses) to effectively model it again in the synthetic. Joshi et al. [19] identify that for the SPEC CPU2000 workloads, most of the load and store instructions have a dominant stride based memory access. We also observe a similar behavior in the SPEC CPU2006 workloads. Figure 5(a) shows the breakdown of the stride access patterns at a granularity of a 64Byte block by different load/store instructions binned into categories 0, 1-3, 4-7, 8-15, 16-31 and $>$ 31. This way of characterizing and portraying the stride access patterns in terms of 64 byte block sizes is similar to the previous work as in Joshi et al. [19] for SPEC CPU2000. It is clearly evident that most of the benchmarks exhibit a stride based behavior. Benchmarks 456.hmmer, 473.astar, 436.cactusADM have a dominant stride of zero for more than 90% of the memory accesses meaning that 90% of their accesses are within the same 64 byte block and will probably result in a lot of hits in the cache due to spatial locality. Totally, 12 benchmarks out of the 22 benchmarks studied have dominant stride for more than 75% of the memory accesses. For the ImplantBench suite, Figure 5(b) shows the break down of the stride at a 64 byte block granularity and it can be observed that the ImplantBench suite also has the same dominant stride behavior as CPU2006 and CPU2000 suites.

Capturing Memory Level Parallelism Information: Memory Level Parallelism information of the workloads is captured and the Figures 6(a) and 6(b) show the distribution of number of outstanding long-latency loads as box plots for the different workloads in the CPU2006 and ImplantBench suites respectively. By using the stride information, the synthetics mimic the hit/miss rate behavior of the original workloads. The impact of these hit/miss rates on the execution time is taken care of by capturing the memory level parallelism information. It can be noted that 483.xalancbmk, 410.bwaves, 436.cactusADM, 433.milc, 437.leslie3d, 459.GemsFDTD, 462.libquantum have relatively higher amounts of MLP compared to other benchmarks. It is to be noted that most of these benchmarks are floating point benchmarks except 462.libquantum and 483.xalancbmk. We also record the number of consecutive dynamic instructions when there are no outstanding long-latency loads to model the frequency of the bursty misses.

Also, to match the MLP of the original workloads, recording and modeling the load-to-load dependencies that exist in the original application plays a significant role. When a long-latency load is dependent on another long-latency load, there cannot be any overlap in execution between these loads. The previous synthetic benchmark generation approaches modeled the dependency distances only at the detail of the consumer instruction type and did not record the type of the producer. Our experimental results (Section 4) show that, at least this information has to be captured for load instructions to be able to match the memory behavior of some of the modern workloads.

In the rest of the paper, we explain how we use the profiled data of the benchmarks to generate the synthetic clones that will have a similar behavior as the original. We then compare both the microarchitecture dependent and microarchitecture-independent characteristics of the synthetic benchmark to that

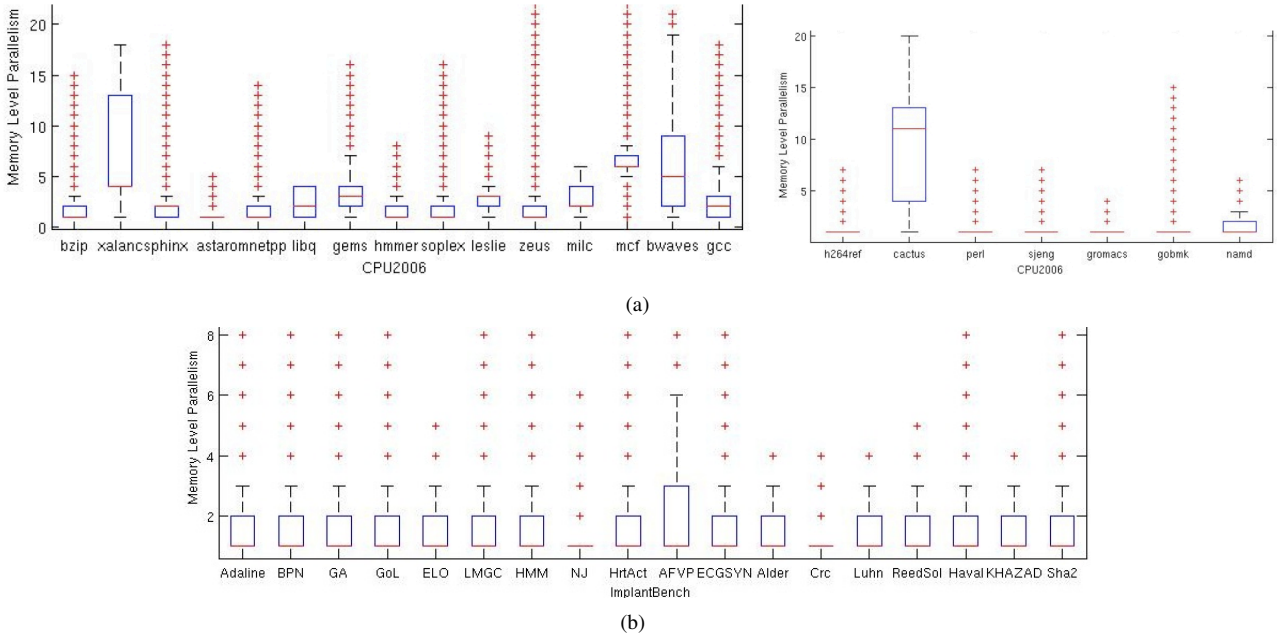


Fig. 6. Captured MLP information as box plots showing the distribution of the burstiness of long-latency loads for (a) CPU2006 workloads (b) ImplantBench workloads

of the original workload.

B. Code Generation Framework

The generated synthetic program will be a C code with a header and an outer wrapper loop with a set of two inner loops, each populated with embedded assembly instructions. The number of iterations of the outer most wrapper loop controls the number of dynamic instructions in the generated synthetic. The number of times each of the inner loops are iterated controls the data footprint of the generated synthetic. Each of these inner loops are similar in everything except their memory access model. The header part of the C code consists of declaration and memory allocation for the data structures, which are primarily a set of single dimensional arrays. 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. [5], but with an improved memory access, branching and ILP models. Our code generation algorithm uses the information captured in the profiling phase (as in Figure 2(b)) and synthesizes code for each of these inner loop as follows,

1) Based on the instruction footprint (SFG information) of the original workload, the number of basic blocks to be included in each of the inner most loops is determined.
2) For each basic block to be generated, the instruction pattern (in terms of instruction type - int, float etc) is randomly chosen from the pool of instruction patterns (of the basic blocks) in the original workload. The execution frequency of each of the patterns in the original workload governs the number of times that particular pattern is chosen to be included in the synthetic.
3) If detailed instruction pattern information for the basic blocks are not available, using the mean, standard deviation of basic block size and the overall global instruction mix information is used to populate the instructions into the basic blocks.
4) The basic blocks are bound together by using conditional branches at the end of each of the basic block. Our branching model is explained in detail in the later part of this Section.
5) Using the dependency distance distribution for each of the instruction types, each instruction in each basic block is assigned a producer instruction for each of its operands within the loop. If these producer-consumer instructions are not compatible with each other, the algorithm moves above/below, one or more instructions until it finds a matching producer for each instruction.
6) Destination registers are assigned to each of the instructions in a round robin fashion using the pool of available registers. Based on the previously assigned dependencies, the registers are assigned to various input operands.
7) Each of the load/store instructions is configured based on the memory model explained in Subsection 2 of this Section.
8) The number of times this particular inner loop has to be iterated is decided based on the data foot print of the original workload as explained in the memory model in Subsection 2.
9) 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.

1) Transition Rate Based Branch Behavior: Some previous synthetic benchmark generation approaches [5] used the branch misprediction rate of the different branches of the original workloads to mimic the branching behavior. In our case, instead of using the branch predictability, we use the microarchitecture-independent metric, branch transition rate [18] as used by Joshi et. al [19] to make the model more robust to microarchitecture changes. 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 transition rate of the corresponding static branch in the original workload. Those transition rates of the branch instructions are grouped into a few pools and each of them is assigned a register. This register is incremented every iteration of the loop and a modulo operation is used to decide if a branch is taken or not taken to mimic the transition rate of that particular pool of branches.

2) Stride Based Memory Access Behavior: Our stride based memory access model is similar to that of the model used by Joshi, et al. [4], except for the fact that our model is more robust as we also use the MLP information of the workloads along with the stride information to generate the synthetic. Each of the static loads and stores in the synthetic benchmark walk one of the allocated memory arrays in a constant strided pattern until the required data foot print of the application is touched and after which, they again start from the beginning of the array. The other integer ALU instructions in the generated synthetic are used to perform the address calculation for these loads/stores. Since we have a limitation on the number of registers that we can use to hold addresses for these loads/store instructions, we cluster the different stride values of the loads/stores of the original program and assign only one register for each of these clusters. A representative stride value for each of the clusters is chosen. The integer ALU address calculation instructions add the stride value of the particular cluster to the allocated register and write the sum (new address) back into the same register. The loads/stores are synthesized in such a way that they access the address in this register assigned to their corresponding cluster.

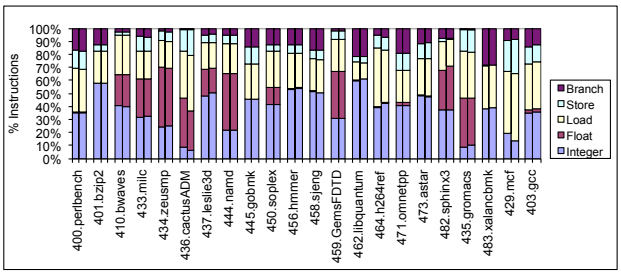
3) Model for the Memory Level Parallelism: During the synthesis of the clone, we can achieve the desired MLP in the synthetic by having control over the following: 1) the placement of highly strided loads (closer to each other or farther from each other) 2) the cluster to which these loads get assigned (i.e., whether they use the same base register or not) and 3) the number of load-load dependencies 4) Usage of more than one loop in the synthetic. The highly strided loads are the long-latency loads which miss in the last level of the cache. By grouping these long-latency loads, MLP of the synthetic can be controlled. When two loads share the same base register, the possibility of the second load accessing the same block as that of the first load is high and hence, it has an impact on the MLP of the synthetic. When a load is dependent on another load, these two load instructions cannot be outstanding misses at the same time and thus, this also controls the amount of MLP in the synthetic. Though the first two of the above said techniques are relatively easier to implement, it is not trivial to make a load instruction dependent on another load in our memory model due to walking an uninitialized memory array. If such a dependency is assigned directly, the 'consumer' load could access an invalid memory location. Initializing the memory array in the header of the synthetic alters the locality behavior of the synthetic. These special dependencies are handled as indirect dependencies in our framework by introducing an existing ALU instruction in the dependence chain to ensure the access to a valid memory location.

Also, some modern workloads have multiple loops with varying data access behavior. Say, for example a program has two loops, one loop to initialize a large data structure and another loop to perform a set of computations using this data structure. Such a program will see a lot of misses in its first initialization loop and when the data structure fits into the cache, the second execution loop exhibits a good locality. Modeling such a benchmark in the synthetic with only one loop and limited instruction footprint is not possible. We need at least two loops to model memory behaviors where there are bursty misses and big execution intervals without any misses. This is the reason for capturing the metric of "mean number of consecutive dynamic instructions when there are no outstanding long-latency loads". The MLP metric only indicates the burstiness of long-latency loads when there is at least one outstanding long-latency load. But, the other metric also explains how frequently these bursty behaviors happen. Since this kind of initialization-execution behavior happens only in a few of our target workloads, in our framework, we analyze this behavior manually and hand tune the number of iterations of the first and the second loops based on the metric defined above for these workloads. The number of iterations of the second loop is determined based on the average interval of execution in terms of number of instructions for which there are no outstanding long-latency loads. The first inner loop in the synthetic can be used like an initialization phase and the second as the execution phase (only if required). Most of the workloads that we have characterized can be modeled with only one loop with an effective usage of the first three techniques to model MLP. There are a few workloads like 400.perlbench, 435.gromacs, 444.namd, 445.gobmk, that have intervals of thousands of dynamic instructions without any long-latency loads and, thus, require the usage of two loops. Since the characteristics of the programs are statistically modeled, most of the fine-grained phase behavior gets averaged out in the synthetic. For very coarse-grained phase behavior, use of two loops, as described above helps to approximately model this behavior.

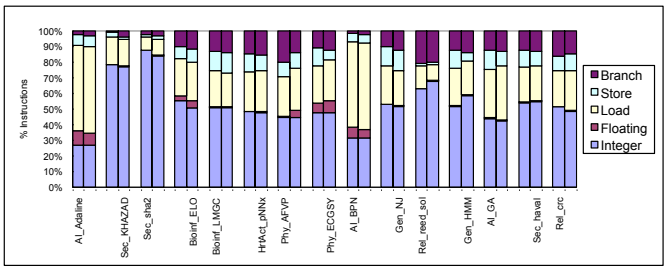
IV. Results and Analysis

A. Accuracy in the representativeness of the synthetic clones

The accuracies of the synthetic benchmarks in capturing the performance of the original application is evaluated by



(a) SPEC CPU2006



(b) ImplantBench

Fig. 8. Comparison of the Instruction mix of the original and the synthetic workloads. Amongst the two bars shown for each benchmark, the bar on the left is for the original and that on the right is for the synthetic

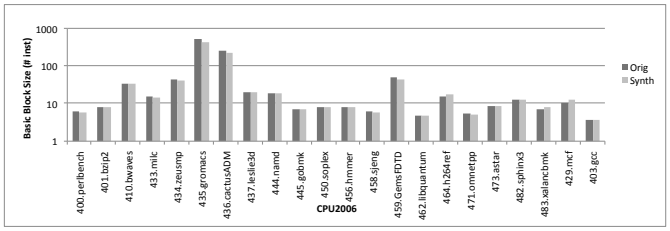


Fig. 7. Comparison of the basic block size between the synthetic and the original workloads for CPU2006

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

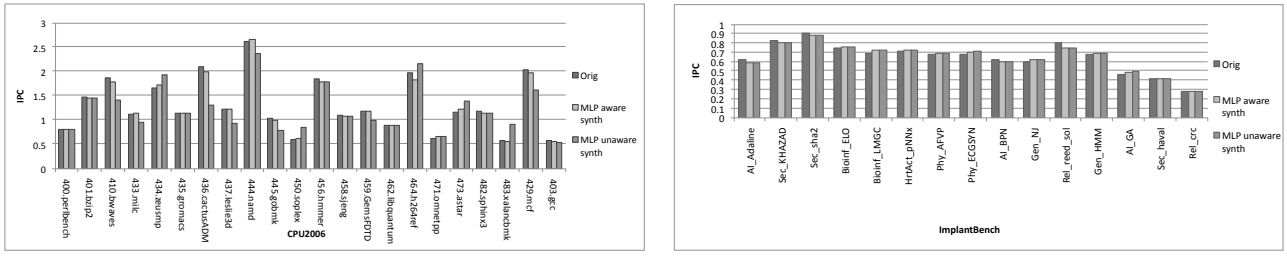
Fig. 9. Machine configurations used: Machine-A for SPEC CPU2006 and Machine-B for ImplantBench workloads

comparing both microarchitecture dependent and independent metrics. First, we compare the microarchitecture-independent metrics like the basic block size, instruction mix, and dependency distance distribution of the original to that of the synthetic. Figure 7 shows the arithmetic mean of the basic block sizes of the original and the synthetic in the logarithmic scale. These arithmetic means were calculated based on the number of instructions in the basic blocks and the dynamic frequency of execution of each of the basic blocks. It can be observed that the basic block sizes of the synthetic match the basic block sizes of the original with an average error of 3.9%.

The Figures 8(a) and 8(b) show the instruction mix of the synthetic benchmark and that of the original benchmarks for the CPU2006 and ImplantBench workloads. It can be found that the instruction mix of the synthetic matches that of the original very closely and the average errors are within 5%. Even the minimal error in the instruction mix occurs when the effective address calculation for loads/stores or the modulo operation for the branch needs to be done and there are not enough integer ALU instructions in the original benchmark. The dependency distances of the original and the synthetic are compared based on each of the instruction type and the error is evaluated. Usually the dependency assigning algorithm does not have to move up/down more than 2-3 instructions before it successfully assigns the dependency for our target workloads. While the average error in dependency distances for various types is within 7%, the main source of the error is the first operand of the integer ALU operations. This is due to the changes in the dependency distances that happen when an integer ALU instruction is made as a load/store effective address calculation instruction. In that case, the original dependency distance of the integer instruction is overridden by the distance from the producer of the base address.

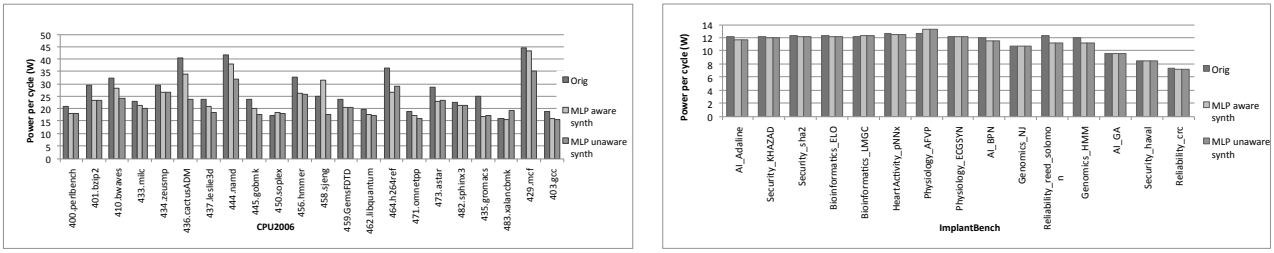
The execution time and power consumption of a benchmark are the first class performance metrics used in computer architecture to assess the performance of a benchmark on a processor/system. Since, we aim at miniaturization of the workloads in terms of the execution time, Instruction-Per-Cycle (IPC) and power-per-cycle are the metrics that we have used to compare the performance of the original and the synthetic workloads to show the efficacy of the generated synthetics. We determine both the accuracies of using the synthetics as proxies to evaluate the performance of a given microarchitecture and the sensitivity to various micro-architectural design changes. To evaluate the performance of the CPU2006 workloads, we simulate both the original and the synthetic on the simoutorder simulator of SimpleScalar [2] for a typical modern machine configuration (Machine-A) given in the figure 9. For the experiments on ImplantBench, we use a typical configuration of an embedded processor (Machine-B) as given in Figure 9. These machine configurations are the same as used in some previously published work [19].

As shown in Figure 10(a), the synthetics for CPU2006 have an average error of 2.8% and a maximum error of 7.7% for the benchmark 464.h264ref when using the MLP information in the synthetics. While using the previous synthetic generation methodologies (without MLP information) as in previous work [5] [4], the average error in IPC is 15.3% clearly showing the



(a) SPEC CPU2006 (b) ImplantBench

Fig. 10. Comparison of IPC between the synthetic and the original workloads



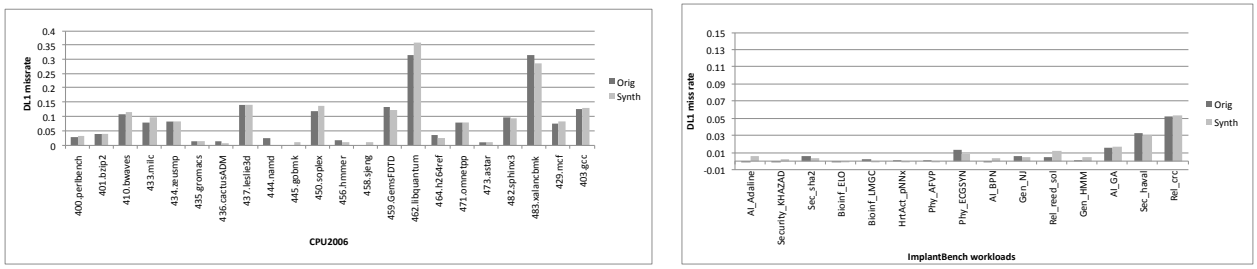
(a) SPEC CPU2006 (b) ImplantBench

Fig. 11. Comparison of power-per-cycle between the synthetic and the original workloads

importance of an MLP aware workload generation. It should be noted that the benchmarks 410.bwaves, 456.cactusADM and 483.xalancbmk that have high MLP as shown in Figure 6(a) have decreased error rates while using our MLP aware synthetic clone generation. The importance of modeling the type of the producer instruction for a consumer load instruction while modeling the dependency distances can be explained with 450.soplex as an example. The benchmark 450.soplex solves a linear program using a simplex algorithm and sparse linear algebra and it has a lot of load instructions that are dependent on other load instructions. When this load-load dependency information is not incorporated into the synthetic, this benchmark results in 40% error compared to 2.7% when using this information. For IPC and power results, 15 benchmarks benefit from the 3 automated MLP techniques, 4 benchmarks benefit from all the 4 MLP techniques (with two loops) and 6 benchmarks do not benefit from the MLP techniques. There is an error of 4.43% in IPC when only automatic MLP techniques are used as opposed to 15.3% for MLPunaware. The usage of two loops with manual intervention reduces the IPC error further to 2.8%. The accuracies in the IPCs of the ImplantBench suites are given in Figure 10(b). The average IPC error for the workloads in the ImplantBench suite is 2.9% and a maximum error of 7.2%.

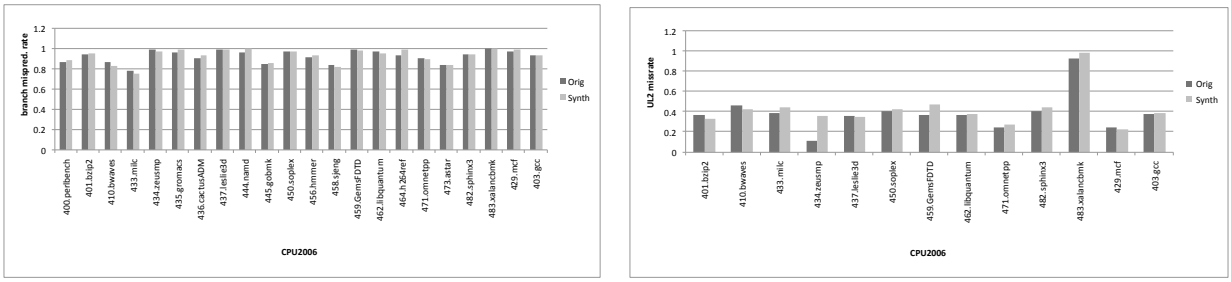
To evaluate the power consumption of the synthetic and the original workloads, we use the Wattch [20] simulator extension of the SimpleScalar tool set. We use the most aggressive clock gating setting in Wattch and compare the average power consumption per cycle of the synthetic and that of the original workload. Figure 11(a) shows this comparison for CPU2006 and it is to be noted that the average error in power per cycle is 14% and the maximum error is 33% for the benchmark 435.gromacs. The average size of basic blocks in this benchmark is 512 instructions in the original and when we try to miniaturize the benchmark based on the execution frequencies of the basic blocks, we lose some long basic blocks that have a significant impact on the power characteristics. If a user is more concerned about the errors in these benchmarks being so high, the only solution is to compromise on the speedup to achieve higher accuracies by including more basic blocks into the synthetic. The other significant source of error in power-per-cycle for the remaining benchmarks is due to the fact that the long running original applications have higher power consumption in the instruction cache than these relatively very small synthetic clones. It can be observed that power consumption is mostly underestimated by the synthetic, bringing up the possibility of correcting it. For the ImplantBench suite, the average error in power consumption is 2.5% and a maximum error of 9.2% which can be seen in Figure 11(b). This error is less than CPU2006, since these workloads have relatively lower dynamic number of instructions.

Figures 12(a) and 13(a) show the error in the miss rates in the Data Level 1 (DL1) cache and the branch misprediction rates of the synthetic compared to the original workload for SPEC CPU2006. The average error in the DL1 hit rate for CPU2006 is 1.06% and that in the branch predictability is 1.7%. The DL1 miss rate comparison for ImplantBench is shown in Figure 12(b). Most of the ImplantBench workloads being simple have very low L2 miss rates and high branch predictability. Thus, we only show the accuracies in L2 cache miss rates and branch predictability of the SPEC CPU2006 workloads. Figure 13(b) shows the error the miss rate in the Unified Level 2 (UL2) cache compared to the originals for those benchmarks that have at least more than 3% of the DL1 accesses reaching



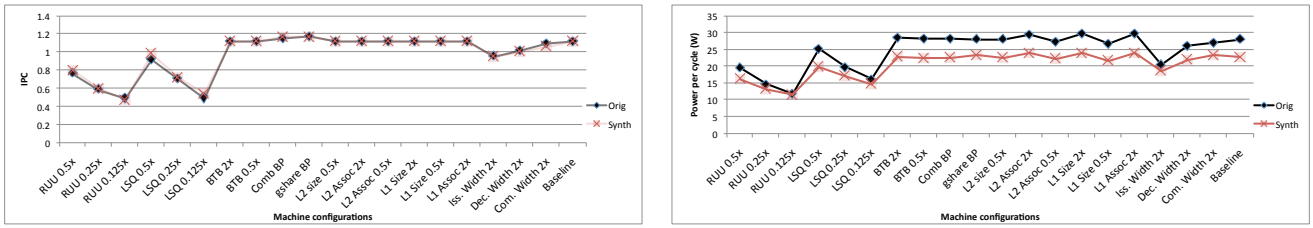
(a) SPEC CPU2006 (b) ImplantBench

Fig. 12. Comparison of DL1 miss rate between the synthetic and the original workloads



(a) Branch misprediction rate comparison (b) UL2 miss rate comparison

Fig. 13. Comparison of UL2 miss rate and branch misprediction rate between the synthetic and the original workloads for CPU2006



(a) IPC (b) Power-per-cycle

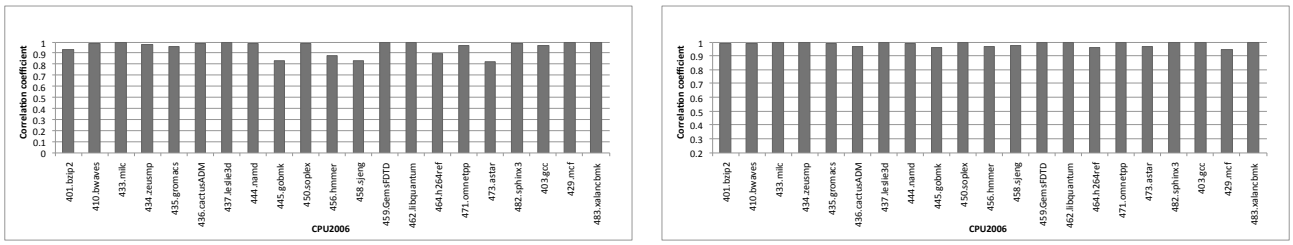
Fig. 14. Comparison of the variation of IPC and power-per-cycle for 433.milc between the synthetic and the original

UL2. When the number of UL2 accesses are too small, the impact of the accuracy in UL2 miss rates on IPC is also small. The benchmark 434.zeusmp has a high error in the UL2 miss rate compared to the original benchmark. It is a computational fluid dynamics application that is used for the simulation of astrophysical phenomena. This benchmark has a very large data footprint compared to any of the other benchmarks that we have used in this study. It has an almost 1GB of data foot print for the top 100 million instruction simulation point. This benchmark has a miss rate of 8.5% in the DL1 and has a miss rate of only 10% in the UL2. A very detailed modeling of the working set size at a much smaller granularity in terms of the dynamic execution interval is required for this benchmark to capture its overall memory access behavior more precisely than what is dealt with, in this paper. We do not show the error rates in the Instruction Level 1 cache because we found that the number of misses is very small for a typical modern processor configuration.

B. Accuracy in the sensitivity to design changes

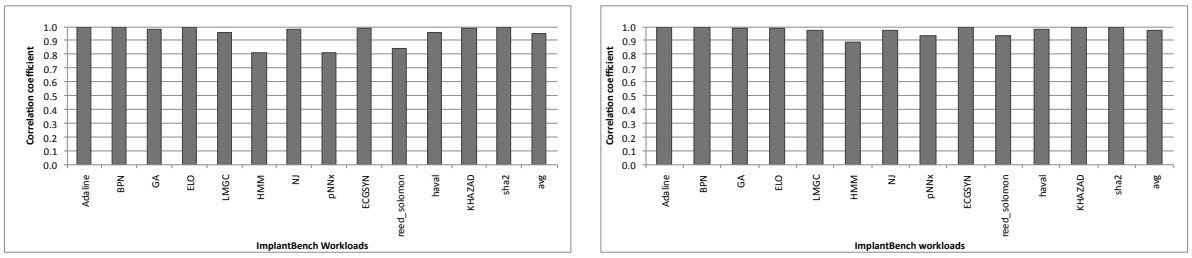
In an architecture study, the accuracy in assessing the performance impact of design changes [21] [22] is more important than assessing the performance for a particular microarchitecture. We evaluate the synthetics to see the sensitivity to various design changes. We study accuracies for changes in the size of the Register Update Unit (RUU), Load Store Queue (LSQ), Branch Target Buffer (BTB), the type of the branch predictor used, size of the Unified L2 cache, Unified L2 associativity, Data L1 cache size, Data L1 associativity, issue width, decode width and the commit width of the machine. The IPC and power variation for the CPU2006 floating point benchmark 433.milc to design changes are given in the Figures 14(a) and 14(b) respectively. 433.milc is one of the benchmarks that is very sensitive to the different design changes under study.

The correlation coefficients in IPC between the synthetic and the original for the set of 19 design points as used in Figure 14(a) are shown in the Figure 15(a) for CPU2006 workloads. The correlation coefficient is directly proportional to correctness of the synthetic in following the trends of the



(a) IPC (b) Power-per-cycle

Fig. 15. Correlation coefficient for IPC and power-per-cycle between the synthetic and original for different machine configurations for SPEC CPU2006



(a) IPC (b) Power-per-cycle

Fig. 16. Correlation coefficient between the IPC and power-per-cycle of the synthetic and the original for different machine configurations for ImplantBench

original for the different design points. The average of the correlation coefficient for IPC is 0.95 for CPU2006. Similarly, Figure 15(b) shows the correlation coefficients of the synthetic with the original in assessing the power per cycle metric for CPU2006 workloads. The average correlation coefficient for power is 0.98 for CPU2006. Figures 16(a) and 16(b) show the correlation coefficients for IPC and power consumption for the ImplantBench workloads. The average correlation coefficient for IPC is 0.94 and that for power-per-cycle is 0.97 for the ImplantBench workloads.

C. Cloning selected full runs of CPU2006

Previously, we have shown the efficacy of our synthetic benchmark generation methodology by cloning the top simulation point of the different workloads in the SPEC CPU2006 suite. This was due to the prohibitive simulation time that is required to profile the CPU2006 workloads completely for various machine configurations used in this study. To bring out the effectiveness of the methodology for cloning complete runs, we have profiled the complete runs of six workloads that have relatively less simulation time than others and generated synthetics for these workloads. We have compared the performance of these synthetics with the original complete run in terms of the IPC. Figure 17(a) shows the IPC comparison results. The average error is 3.74% in IPC. The table in Figure 17(b) shows the dynamic number of instructions in the full run and that of the synthetic and the speedup that is achieved. An average speedup of 5 million (in terms of instructions) is achieved for the six selected workloads.

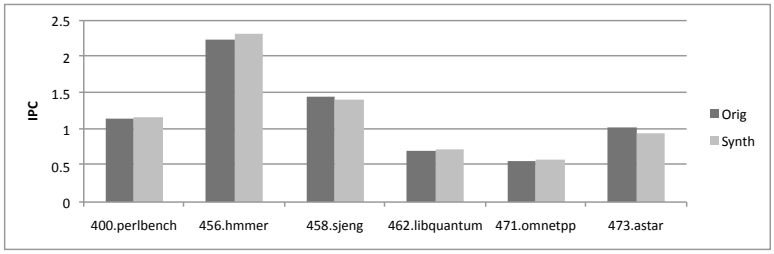
Our methodology is found to be superior in both accuracy and miniaturization compared to simulation points. For the 100 Million instruction simpoints used in the study [3], the average error when using all the simulation points (generated with max number of simpoints=30) is around 5%. If, say a typical benchmark had 15 simulation points, the number of dynamic instructions simulated will be 1500 million instructions. It is very common to use only one simulation point and the error should be much higher when only one simulation point is used. Rather our methodology gives only an error of 3.7% for synthetics of length less than a million instructions. This could be attributed to the reason that these synthetic instruction sequences are constructed based on characteristics of the whole program, rather simpoint methodology is forced to leave some characteristics to be able to choose one contiguous dynamic instruction chunk.

V. Limitations

The provided synthetics are intended to be used by designers, who honestly want to explore the design space in early design stage of a processor. The synthetics should not be used as a sole method to publish final performance numbers. Other methods should be used after narrowing down the design space. These synthetics are not meant to be further optimized by the compiler, because, we have incorporated some characteristics that should be exhibited by the synthetic already using embedded assembly.

VI. Summary

We have characterized the SPEC CPU2006 workloads mostly based on microarchitecture-independent characteristics and have formulated and provided miniaturized synthetic clones [6] for these workloads to aid in accelerating architecture simulations with simulation speedups of up to 6 orders of magnitude. Along with that, we also provide the absolute and the relative accuracies of these synthetics in predicting the performance



(a) IPC comparison

Benchmark	# of Instns in billions (original)	# of Instns in millions (synthetic)	Speedup
400.perlbench	184.5	0.19	936238
456.hmmer	2593.1	0.29	8724843
458.sjeng	3187.7	0.30	10357323
462.libquantum	1989.0	0.56	3495214
471.omnetpp	730.0	0.12	5692522
473.astar	966.5	0.25	3830291

(b) Dynamic number of instructions and speedup

Fig. 17. IPC comparison and speedup information for complete runs of some CPU2006 workloads

and the power consumption of various microarchitectures. We compare our MLP aware synthetic benchmark generation methodology with previous approaches [4] [5] and show that the synthetic benchmarks generated using our methodology have 12.5% more accuracy in terms of IPC in the representativeness of the synthetics to that of the original workloads. The synthetics generated using our methodology have a correlation coefficient of 0.95 and 0.98 for IPC and power-per-cycle for the sensitivity to changes in microarchitecture. The availability of the provided synthetic clones will enable computer architects to use these latest workloads instead of the older SPEC suites for future studies. We have also characterized and provided the synthetic clones for the futuristic workloads to be used in bio-implantable systems.

VII. Acknowledgements

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 and 0751091, Lockheed Martin, Sun Microsystems and IBM. 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] Standard Performance Evaluation Corporation http://www.spec.org.

[2] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342. *University of Wisconsin, Madison*, June 1997.

[3] Karthik Ganesan, Deepak Panwar, and Lizy 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*.

[4] Ajay Joshi, Lieven Eeckhout, Robert H.Bell Jr., and Lizy K. John. Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks. *International Symposium on Workload Characterization. (IISWC)*, October 2006.

[5] Robert H. Bell Jr., Rajiv R. Bhatia, Lizy K. John, Jeff Stuecheli, John Griswell, Paul Tu, Louis Capps, Anton Blanchard, and Ravel Thai. Automatic Testcase Synthesis and Performance Model Validation for High Performance PowerPC Processors. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2006.

[6] http://lca.ece.utexas.edu/tools.html.

[7] Greg Hamerly, Erez Perelman, and Brad Calder. How to Use SimPoint to Pick Simulation Points. *ACM SIGMETRICS Performance Evaluation Review*, March 2004.

[8] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. *Proceedings of the International Symposium on Computer Architecture, (ISCA 2003), p. 84 - 95*.

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

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

[11] Sebastien Nussbaum and James E. Smith. Modeling Superscalar Processors via Statistical Simulation. *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[12] Lieven Eeckhout, Robert H. Bell Jr., Bastiaan Stougie, Koen De Bosschere, and Lizy 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.

[13] Wing Shing Wong and Robert J. T. Morris. Benchmark Synthesis Using the LRU Cache Hit Function. *IEEE Transactions on Computers*, 1988.

[14] Cheng-Ta Hsieh and M. Pedram. Microprocessor power estimation using profile-driven program synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, November 1998.

[15] E.S. Sorenson and J.K. Flanagan. Evaluating synthetic trace models using locality surfaces. *2002. WWC-5. 2002 IEEE International Workshop on Workload Characterization*, November 2002.

[16] Stijn Eyerman and Lieven Eeckhout. A Memory-Level Parallelism Aware Fetch Policy for SMT Processors. *IEEE 13th International Symposium on High Performance Computer Architecture, 2007. (HPCA)*, February 2007.

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

[18] Haungs M, Sallee P, and Farrens M. Branch Transition Tate: A New Metric For Improved Branch Classification Analysis. *Sixth International Symposium on High-Performance Computer Architecture (HPCA) Page(s):241 - 25*, January 2000.

[19] Ajay Joshi, Lieven Eeckhout, Jr. Robert H. Bell, and Lizy K. John. Distilling the Essence Of Proprietary Workloads Into Miniature Benchmarks. *ACM Transactions on Architecture and Code Optimization (TACO)*, August 2008.

[20] Margaret Martonosi, Vivek Tiwari, and David Brooks. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. *isca, pp.83, 27th Annual International Symposium on Computer Architecture (ISCA 2000)*.

[21] Engin Ipek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, Oct 2006.

[22] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, Oct 2006.