Workload Synthesis for a Communications SoC

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

Abstract – Communications processors are SoCs that contain multiple processor cores, an on-chip network, memory controllers, frame managers, encryption and pattern matching engine, etc. In early design exploration of such SoCs, it is important to have workloads as close as the final real-world workloads that the SoC will be running. However at the same time, it is often difficult to run such realistic workloads on early performance models and simulators. Under such scenarios, it is extremely useful to be able to have simplified workloads that are similar to the realistic workloads.

In our recent research, we have created a benchmark synthesis process, which consists of synthesizing a proxy workload that possesses approximately the same performance and power characteristics as the original workload. The synthesis comprises of two steps: (1) profiling the real-world proprietary workload to measure its inherent behavior characteristics, and (2) modeling the measured workload attributes into a synthetic benchmark program. The set of workload characteristics can be thought of as a signature that uniquely describes the workload’s inherent behavior, independent of the microarchitecture. The cloned code in fact has no functionality and cannot be reverse engineered to create the original code or algorithms. The cloned software can be freely released to hardware developers so that they can optimize the hardware they deliver to their clients to yield improved performance. This paper describes the benchmark synthesis process as well as the status of the research.

An automated synthetic benchmark generation methodology is presented first as an approach to synthesize benchmarks for early performance studies from real-world applications. The process of constructing synthetic proxies that approximately resemble the original application is explained. Experiments with generating proxy workloads for the Freescale QorIQ Communications SoC P4080 are presented.

Keywords – Synthetic benchmark; SoC benchmark; Stressmark

I. INTRODUCTION

A System on a Chip contains multiple components often possessing computation and communication capabilities integrated on a chip. The common features and requirements of these platforms include the need for rich and diverse functionality, high performance and low power consumption. Designing and evaluating these platforms in a timely manner, ensuring the demanded performance and power efficiency is of utmost importance. System level benchmarks covering the diverse functionality are rare.

Communications processors are SoCs that contain multiple processor cores, an on-chip network, memory controllers, frame managers, encryption and pattern matching engine, etc. As an example, consider the QorIQ P4080 Communications processor from Freescale illustrated in Figure 1. In early design exploration of such SoCs, it is important to have workloads as close as the final real-world workloads that the SoC will be running. However at the same time, it is often difficult to run such realistic workloads on early performance models and simulators. Under such scenarios, it is extremely useful to be able to have simplified workloads that are similar to the realistic workloads.

Figure 1. Block diagram of QorIQ P4080 processor[1]

In our recent research [2-3][7-11], we have created a benchmark synthesis process, which consists of synthesizing a proxy workload that possesses approximately the same performance and power characteristics as the original workload. The synthesis comprises of two steps: (1) profiling the real-world proprietary workload to measure its inherent behavior characteristics, and (2) modeling the measured workload attributes into a synthetic benchmark program. The set of workload characteristics can be thought of as a signature that uniquely describes the workload’s inherent behavior, independent of the microarchitecture. The cloned code in fact has no functionality and cannot be reverse engineered to create the original code or algorithms. The cloned software can be freely released to hardware developers so that they can optimize the hardware they deliver to their clients to yield improved performance. This paper describes the benchmark synthesis process as well as the status of the research.

An automated synthetic benchmark generation methodology is presented first as an approach to synthesize benchmarks for early performance studies from real-world applications. The process of constructing synthetic proxies that approximately resemble the original application is explained. Experiments with generating proxy workloads for the Freescale QorIQ Communications SoC P4080 are presented.

Keywords – Synthetic benchmark; SoC benchmark; Stressmark
applications. The process of constructing synthetic proxies that approximately resemble the original application is explained. Experiments with generating proxy workloads are presented using the Freescale QorIQ Communications SoC P4080 for validation.

The automatic workload synthesis can help SoC design in multiple ways: (i) generation of system level benchmarks covering multiple units (ii) system level power and thermal stressmarks covering the various units (iii) system level fixed power benchmarks fitting a power budget. In this paper, we describe methodology to attain these goals.

The rest of this paper is organized as follows. We provide related work in Section 2. In Sections 3 and 4, we present an overview of system level SoC synthetics and system level SoC power stressmarks respectively. The QorIQ communication P4080 experimental platform is explained in the Section 5. In Section 6, we explain the detailed methodology of the synthetic cloning and stressmark generation. Preliminary results for both applications are shown. Lastly, section 7 concludes the paper.

II. RELATED WORK

The most commonly used techniques to reduce simulation time are sampling techniques such as Simpoint [6,14-16]. However, these techniques require either fast-forwarding support from the simulator or huge checkpoint files to reproduce the output. The problem is that it is very inefficient to use fast-forwarding when the interval of execution of interest is located in the later stage of the program execution. Also, checkpoint file requires huge storage space and it is hard to distribute to others. On the other hand, the synthetic benchmark approach provides very small size source code/execution file which is very efficient in run time and storage space.

Another approach to reduce the simulation time is benchmark subsetting [17] which selectively run a subset of benchmark suite whose characteristics are representative of the whole set. This approach is useful when hardware is ready and benchmarking can be finished in a short time. However, in pre-silicon design stage, it is impractical to run even the subset of the benchmark since the full runs of selected programs are too big to directly run on simulators.

The idea of using statistical simulation to guide the design space exploration was introduced [18, 19]. Eeckhout et al. [20] used the execution frequency of the basic blocks and their transition probability to characterize the control flow behavior of a program. Wong et al. [21] proposed synthesizing benchmark by using the profile of the workload. Joshi et al. [10] proposed creating synthetic benchmarks with microarchitecture independent characteristics. Synthetic benchmarks were generated in embedded assembly format to precisely control the performance.

III. SYSTEM LEVEL SOC SYNTHETICS

In early design exploration of SoCs, it is important to have workloads as close as the final real-world workloads that the SoC will be running. However at the same time, it is often difficult to run such realistic workloads on early performance models and simulators. Early models of such systems are in a RTL or in a high-level language such as C and are very slow. They cannot run true system level real-world applications. However, it is important and extremely useful to be able to have simplified workloads that are similar to the realistic workloads.

The system level synthetics are proxy workloads that possess approximately the same performance and power characteristics as original, complex and often proprietary workloads. Figure 2 illustrates the two synthesis steps: (1) profiling the real-world proprietary workloads to measure their inherent behavior characteristics, and (2) modeling the measured workload attributes into a synthetic benchmark program. The set of workload characteristics can be thought of as a signature that uniquely describes the workload’s inherent behavior, independent of the microarchitecture. The cloned code in fact has no functionality and cannot be reverse engineered to create the original code or the involved algorithms. The cloned software can be freely released to hardware developers so that they can optimize the hardware they deliver to their clients to yield improved performance.

IV. SYSTEM LEVEL SOC POWER STRESSMARKS

SoCs work with stringent power constraints and it is necessary to appropriately estimate maximum power conditions on the SoC. In past research we were able to use the workload synthesis framework in conjunction with a power simulator and a genetic algorithm to identify benchmarks that stress the system [9][10]. Previous research [9] focused power consumption on high-end processor core and it did not consider memory system or peripheral devices. Another research [10] addressed memory hierarchy on the high-end processor but it
did not put any effort on the peripherals, either. In this work, we propose the creation of SoC power stressmarks to maximize the power consumption on both the processing cores and communication/peripheral devices.

An overview of the stressmark generation methodology is shown in Figure 3. At start, the code generator synthesizes code for an initial set of parameter values. The power for the generated code is estimated by an architectural power simulator and results are fed to a genetic algorithm which identifies suitable future sets of parameter values. The process is repeated for several generations of the genetic algorithm until the algorithm maximizes the objective function, in this case, the power of the SoC.

In addition to maximum power benchmarks, the same framework can be modified to create fixed power benchmarks of a certain target power. If some units in an SoC have specific power constraints and operation at a specific power envelope is desired, such a benchmark can be very useful.

V. EXPERIMENTAL PLATFORM

In this section we describe the platform we used for our experiments.

A. Overview

The QorIQ™ P4080 chip from Freescale is a multicore Communications System on a Chip that has eight e500mc Power Architecture® cores, and network and peripheral bus interfaces. Figure 1 provides an overview of the system. The chip is designed to be flexible and it can be used for both compute-intensive and I/O intensive workloads. The processor is designed in 45nm technology with power consumption under 30W.

B. Power Architecture e500mc Cores

The P4080 chip has eight e500mc cores that operate from 533 MHz up to 1.5 GHz. Each e500mc core has 32 KB L1 cache, 32 KB L1 D-cache and 128 KB L2 cache. The processor also has 2 MB of shared front side L3 cache for the tasks that need shared cache.

Each core is capable of running independent operating systems or can run without OS. This feature provides flexible partitioning between the cores that enables more stable operation of the processor. Large number of cores also helps to consolidate multiple functionalities on a single chip.

The e500mc core supports user, supervisor and hypervisor mode of operation that enables virtualization on each core. Virtualization ensures an application to access a system resource only when it is specifically authorized to access. Also, embedded hypervisor enables safe and autonomous operation of multiple individual operating systems, allowing them to share system resources, including processor cores, memory and other on-chip functions.

C. Network Peripherals

CoreNet™ coherency fabric supports both coherent and non-coherent transaction between the peripherals including shared L3 caches. The fabric supports up to 800 Gbps coherent read bandwidth and its prioritization and bandwidth allocation amongst CoreNet end-points eliminates single-point bottleneck between non-competing resources.

Datapath Acceleration Architecture provides packet management, queue management, hardware buffer management, cryptographic security acceleration and regular expression pattern matching. These accelerators enable the processor to perform well in both computation intensive and I/O intensive workloads. Additional I/O peripheral devices are listed in the Table 1.

<table>
<thead>
<tr>
<th>Category</th>
<th>Controller/Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethernet interfaces</td>
<td>Two 10 Gbps Ethernet (XAUI)</td>
</tr>
<tr>
<td></td>
<td>Eight 1 Gbps Ethernet (SGMII)</td>
</tr>
<tr>
<td>High-speed peripheral interfaces</td>
<td>Three PCI Express v2.0</td>
</tr>
<tr>
<td></td>
<td>Two Serial RapidIO® 1.2</td>
</tr>
<tr>
<td>Additional peripheral interfaces</td>
<td>Two USB controllers with ULPI interface to external PHY</td>
</tr>
<tr>
<td></td>
<td>SD/MMC</td>
</tr>
<tr>
<td></td>
<td>SPI controller</td>
</tr>
<tr>
<td></td>
<td>Four I²C controllers</td>
</tr>
<tr>
<td></td>
<td>Two dual UARTs (DUART)</td>
</tr>
<tr>
<td></td>
<td>Enhanced local bus controller (eLBC)</td>
</tr>
<tr>
<td>Other interfaces</td>
<td>Multicore programmable interrupt controller (PIC)</td>
</tr>
<tr>
<td></td>
<td>Two 4-channel DMA engines</td>
</tr>
</tbody>
</table>

VI. METHODOLOGY AND PRELIMINARY RESULTS

A. System Level Synthetics

Synthetic benchmark generation has several steps. First, we profile the desired metrics from the original workload. Based
on the metrics, we generate synthetic benchmarks using assembly inlining of desired instruction sequences. We have developed code generators that create synthetics in Alpha and SPARC ISAs. We do not have a synthesis framework for the Freescale SoC in the Power ISA. Hence we pursued an ISA independent code synthesizer using the compiler’s intermediate representative form [3]. We used Low Level Virtual Machine (LLVM) [10] as our compiler framework to generate synthetic benchmark clones. LLVM is a compiler infrastructure that consists of many modular reusable components that can be built to form a compiler for specific targets. It uses code representation known as LLVM intermediate representation (IR) which is human-readable Static Single Assigned (SSA) format based assembly language. The synthetic is generated in the intermediate form of LLVM and then assembly codes are generated for the target architectures using LLVM compilers. Finally, we compile the synthetic clone and compare the performance with the original. Fig 4 illustrates the flow of this framework.

When one generates synthetics in an intermediate form such as the LLVM IR, it is necessary to avoid further optimizations on the code. The generated code does not have specific functionality, therefore many part of the code are subject to be changed or eliminated during the compiler’s optimization path. Hence LLVM compiler passes were modified to avoid optimizations that would remove dead code or perform other optimizations. The framework in Figure 4 was used to generate synthetics for the Freescale P4080 SoC.

The branch transition rate captures how quickly a branch transits between taken and not-taken paths. It indicates how easy or hard a branch predictor can accurately predict the branch. A branch with a low transition-rate usually has higher branch prediction rate since it switches direction less for a given period of time.

Instruction mix of the original program should be preserved to maintain proper IPC, power consumption and usage of peripheral devices. Unlike generating synthetic clones for the processor, cloning the SoC type of application requires tracking the usage of peripheral devices.

Instruction Level Parallelism (ILP) is a metric to determine the extent to which the pipeline is used waiting for data dependency. We capture average register dependency distance distribution for each type of the instruction. Instructions that have immediate operand are considered having zero dependency distance.

Data locality affects the behavior in various levels of memory hierarchy and it has critical impact on performance of the synthetic benchmark. We capture stride values of each load and store instructions and synthesize ten stride values. Data region is modeled as ten arrays in the synthetic; each stride is used in load and store instructions to access corresponding array to capture the characteristics such as cache hit rate in all the levels of cache.

A key objective of the profiling is to generate the Statistical Flow Graph (SFG) of the program [2]. This statistical flow graph contains information on the size and instruction-mix of the basic blocks, the rate of transitions between the basic blocks etc aggregated in a statistical manner. Figure 5 illustrates an example statistical flow graph.

### Table II. Metrics Profiled to Characterize the Workloads

<table>
<thead>
<tr>
<th>#</th>
<th>Metric</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dynamic execution frequency of basic blocks</td>
<td>Control flow</td>
</tr>
<tr>
<td>2</td>
<td>Successor information of basic blocks</td>
<td>Predictability</td>
</tr>
<tr>
<td>3</td>
<td>Transition probabilities in SFG</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Average basic block size</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Branch taken rate for each branch</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Instruction pattern in a basic block</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Branch transition rate for each branch</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>% Integer instructions</td>
<td>Instruction mix</td>
</tr>
<tr>
<td>9</td>
<td>% Flating point instructions</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>% Load instructions</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>% Store instructions</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>% Branch instructions</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>% Peripheral control instructions</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Dependency distance distribution per type of instructions</td>
<td>Instruction level Parallelism</td>
</tr>
<tr>
<td>15</td>
<td>Stride value of load and store instructions</td>
<td>Data locality</td>
</tr>
</tbody>
</table>

---

**Profiling the Metrics**

As the first step to capture the characteristics of the original benchmark programs, we measured properties of the workloads which are shown in Table 2. These metrics are categorized into five groups to represent the original program’s run-time behavior. We used Freescale’s Architecture Description Language (ADL) that models e500mc processor as a profiler. ADL model gives functional execution of the program where we can attach a plug-in to get the detailed information of each instruction. Some of the microarchitecture dependant characteristics such as branch prediction rate were measured on Freescale’s QorIQ P4080 processor by using performance counters. P4080 processor does not provide instruction level granularity since we cannot read performance counters for every instruction.
3. Place branch instructions in the end of the basic blocks to bind them together. We group branches by their transition rate and assign each of them with a register. Place a modulo operation on each register to determine whether a branch is taken or not. One of the branch target of the last basic block points to the first basic block so that the whole synthesized blocks form a single loop.

4. Using the dependency distance distribution for each of the instruction types, each instruction in each basic block is assigned with 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 up/down one or more instructions until it finds a matching producer for each instruction.

5. Four to eight arrays with size of 40 MB to 80 MB is created to model data segments of the workload. Each of the load/store instructions is configured to have a stride value and assigned to an array. Higher cache miss rate is modeled as a larger stride value to create larger footprint in a given period.

6. Address generating instruction for each load and store instructions are populated. The addresses of the arrays are incremented by assigned stride value so that the arrays are accessed linearly in execution time.

7. The synthetic code is generated in LLVM IR form which is a Static Single Assignment (SSA) based representation. By using LLVM IR, the synthetic code is not bounded to a specific ISA, but still be able to represent expressions in a higher level language. The generate code can be compiled in various ISAs by using LLVM’s backend compiler.

8. Synthetic code is compiled with modified LLVM’s backend compiler to generate assembly code for targeted ISA. Since the synthetic code does not contain any functionally meaningful code, some results from the instructions are not used. These instructions are eliminated in normal LLVM. However, since we need all the instructions to match the performance of the original, we modified the optimization path in LLVM to generate every instruction we synthesized.

Validation of the Synthetic Clone

After all the aforementioned steps are over, the final output is assembly code for the target architecture. We can generate multiple assembly codes if we are to evaluate the synthetic in multiple platforms. We compile the synthetic assembly file with target architecture’s general compiler. The synthetic binaries are executed in either simulator of the target system or directly on hardware.

The final synthetic clone is configured to have around 300 thousand dynamic instructions which can be run in a few seconds even in performance model simulators. The results in terms of various performance metrics are compared to that of the original workloads.
Figure 6 shows comparison of the normalized IPCs of the original and the synthetic clones [3]. The numbers are normalized to the highest IPC of the original program. This work is done as a primary study to show the efficacy of our framework. The average error is 37.9% with maximum of 212%. The high errors in IPC mainly occur where the originals have very low IPC around 0.2, which is mainly caused by high DL2 misses and memory load dependancies. Total number of DL2 misses is relatively small and their impact on performance is usually minimal. However, some benchmarks have significantly high DL1 miss rate which causes high DL2 miss rate as well. In that case, access latency to the main memory causes significant IPC drop in the workload, which are not accurately captured in the synthetics. IPC of the synthetic benchmarks are higher than the original for those kinds of workloads.

Generating synthetics for the entire SoC remains as future work. Parameters related to peripherals and communication blocks should be incorporated into the parameterized model and code generator. However, the preliminary experiments with the computation components on the P4080 are very encouraging. Use of LLVM appears to be a feasible approach rather than generating the synthetic directly in each machine’s ISA.

### B. System Level Stressmarks

**Abstract Workload Model**

Workload space for the stressmark generation consists of 17 dimensions that falls under four categories. Abstract work model categories are basically the same as synthetic code generation with some additional information on memory level parallelism (MLP). However, their values are restricted in some discrete range which we derived from the various benchmark workloads. The ranges are selected from the existing workloads since our target is to create a stressmark that is realistically attainable through the normal operation. Detailed search space and parameter settings are listed in Table 3.

### IBM SNAP – Genetic Algorithm Tool Set

Genetic Algorithm (GA) is one of the machine learning techniques which is known to be very effective with global optimization problem [4]. GA is a particular class of search heuristics that use techniques like mutation, crossover, inheritance and selection to solve optimization problems. A population in the genetic space is referred to as the set of potential candidate solutions.

GA initiates searching with a population of a set of random individuals or chromosomes. In each generation, GA examines fitness value of each individual in the population and evaluates it. Multiple individuals are selected among the population, then the individuals are crossed over or randomly mutated to form a new population for the next generation. The process of evolution described above is repeated until the desired fitness is achieved.

IBM SNAP [13] is a GA tool set which takes bounds for the various parameters in the abstract workload model given by the user as the description of the searching space. Each individual’s fitness is evaluated, which is power consumption on the microarchitecture in our case. Based on the fitness of each of the different individuals, the next generation of individuals is generated using the genetic operations of copy, crossover and mutation.

SNAP provides the following parameters to control how the individuals are chosen for the next generation, i) Mutation rate: number of individuals that should be probabilistically chosen to mutate ii) Reproduction rate: number of individuals that should be probabilistically chosen to copy into new population iii) Elite reproduction rate: number of fittest individual of previous generations that should be copied into new generation iv) Crossover rate: number of individuals probabilistically chosen to serve as parents for point crossover, where a crossover point within a parent is selected and then interchange the two parent chromosomes at this point to produce two new offsprings. v)
Uniform crossover rate: number of individuals probabilistically chosen to serve as parents for uniform crossover. Uniform crossover is the process in which individual bits in the chromosome are compared between two parents and are swapped with a fixed probability of 0.5.

After the workload parameters of the individuals for the next generation are constructed, they are fed to the code generator to generate the synthetic clone. This synthetic clone is automatically compiled and run on the corresponding processor/full-system simulator to evaluate the power consumption, which is used as a feedback to generate the next generation of individuals. SNAP continues to explore the space until the required fitness is reached or until there is no forward progress for a given number of generations.

**Preliminary Results**

We have not yet been able to create a power simulator for the Freescale processor or to instrument it for power. However, the concept of stressmarks has been tested on Alpha/SPARC simulators and x86 hardware. Our system level maximum power stressmark framework SYMPO can automatically generate stressmarks that beat the industry standard torture test MPRIME and assembly-tuned stressmarks such as the cpuburnin, burnk6, burnp6 etc. Figure 7 illustrates the results. Among the SPEC CPU2006 programs, povray and games consume the most power and they are included for comparison as well. The power values are obtained by measurement on an x86 machine.

![Figure 7](image_url)

**Figure 7:** Comparison of system level stressmark (SYMPO) with industry standard power stressmarks on an x86 machine.

The stressmarks were also evaluated on Sparc and Alpha by simulations. Only MPRIME could be used in this study as the other burnin programs are in x86 assembly language. Stressmarks were generated for three microarchitecture configurations on Sparc and Alpha. With SYMPO, we were able to generate programs that exceed the power levels consumed by MPRIME. We are in the process of creating system level power stressmarks for multicore systems.

**VII. CONCLUSIONS**

In this paper, we proposed the usage of synthetic benchmark generation framework on the communication SoC architecture. With preliminary results, we showed the efficacy of the framework to create 1) system level synthetic benchmark clone and 2) system level stressmark. The benchmark clone can provide a miniaturized proxy for the large workload so that the processor designers can easily run it on pre-silicon design models. The system level stressmark, which is automatically generated by genetic algorithm provides practically attainable maximum power consumption across the system. The synthetics can also be used as proxy workloads for proprietary applications.

**ACKNOWLEDGMENT**

This work also has been supported and partially funded by SRC under Task ID 1797.001 and NSF under grant number 0702694. Part of the research is an extension of Jo’s internship project at Freescale. Any opinions, findings, conclusions or recommendations expressed in this material are those of authors and do not necessarily reflect the views of the SRC or other sponsors.

**REFERENCES**


