Fast System Level Benchmarks for Multicore Architectures

Alper Sen*, Gokcehan Kara†, Etem Deniz‡, Smail Niar†
*Department of Computer Engineering, Bogazici University, Istanbul, Turkey 34342
†ISTV2 UVHC, Campus Mont Houy - 59313, Valenciennes Cedex 9, France
‡Department of Computer Engineering, Bogazici University, Istanbul, Turkey 34342

alper.sen@boun.edu.tr, gokcehan.kara@boun.edu.tr, etem.deniz@boun.edu.tr, smail.niar@univ-valenciennes.fr

Abstract—We present a framework that automatically generates system level synthetic benchmarks from traditional benchmarks. Synthetic benchmarks have similar performance behavior as the original benchmarks that they are generated from and they can run faster. Synthetics can also be used as proxies where original applications are not available in source form. In experiments we observe that not only are our system level benchmarks much smaller than the real benchmarks that they are generated from but they are also much faster. For example, when we generate synthetic benchmarks from the well-known multicore benchmark suite, PARSEC, our benchmarks have an average speedup of 149x over PARSEC benchmarks. We also observe that the performance behavior of synthetics have more than 85% similarity to the real benchmarks.

Index Terms—Synthetic benchmarks, SystemC, Parallel patterns, Performance evaluation, Multicore architecture

I. INTRODUCTION

The cost of performance evaluation has been gradually increasing with the increasing complexities of the systems and the benchmarks that run on them. System level approaches, such as virtual platforms, are commonly used to tackle the complexity problem in design automation. There is also a need to develop system level performance evaluation techniques to speed-up the design process and avoid costly redesigns. In system level performance evaluation, abstract system level models of hardware and software components are used with added timing details to provide relative figures for comparing different design options in early design phases. During later design phases accurate techniques are used for more precise estimations, however these techniques require a lot of implementation details. In this paper, we develop techniques that allow us to perform system-level performance evaluation using synthetic benchmarks.

A synthetic benchmark is a small, simple and accurate program that mimics the performance characteristics of an original benchmark that it is derived from. Synthetic benchmarks have multiple advantages over traditional benchmarks in performance evaluation. First, synthetic benchmarks can be used for speeding up early architectural exploration and performance analysis. Although Transaction Level Models (TLM) [1] of hardware are fast, these models are slower when timing details are added for performance studies. Therefore, benchmarks that run on these hardware models need to be fast and synthetic benchmarks can provide this speed by abstracting the functional behavior of the original benchmark.

Second, since developing new benchmarks from scratch in a new domain is a labor-intensive process, automatic synthetic benchmark generation can help relieve this burden. For example, it is desirable to automatically generate system level synthetics for applications in well-known multicore benchmark suites such as PARSEC [2]. This process can essentially create an “equivalent” of these benchmark suites at the system level. Third, synthetic benchmarks can act as a proxy, hence allowing the sharing of proprietary IPs. For example, our synthetics are generated using the binary of the original application (not the source code) and will not have any resemblance to the source code of the original application. Therefore, the use of synthetic benchmarks also avoids disclosing the source code of the application. Design space exploration by a third party may be implemented using only the synthetic code.

The main motivation to use synthetic benchmarks comes from the slow-downs resulting from the co-simulation of SystemC and Pthreads programs since different simulators need to be accurately synchronized in co-simulation. In this scenario, the original application uses the Pthreads library and the synthetic application uses the SystemC library. We obtain a system level synthetic version of an original multithreaded application, say from PARSEC benchmark suite, that can be plugged into an existing SystemC design. This capability is important because multithreaded applications are commonly used in performance evaluation of newly developed architectures including virtual platforms that contain hardware and software components. In these virtual platforms, SystemC is commonly used to describe hardware components. Furthermore, PARSEC benchmarks are large and slow even in a non-co-simulation environment, and this has led to the development of non-system-level synthetic benchmarks from PARSEC suite as described in related work. Hence, rather than doing architectural exploration or performance evaluation of an existing SystemC design using a potentially large and slow Pthreads application in a co-simulation environment, we can accomplish the same performance evaluation using a small and fast synthetic SystemC application that is similar to the Pthreads application and without the need for co-simulation.

We perform experiments on original benchmarks from PARSEC suite. The average speedup of our SystemC synthetics is 149x compared to the original PARSEC benchmarks. All our benchmarks are similar to the original benchmarks on average 85%, where the similarity is defined as the average of
the error percentage between the synthetic and the original using instructions-per-cycle (IPC), cache-miss-rate (CMR), and branch-misprediction-rate (BMR).

Note that our work is orthogonal to the more precise performance estimations that require a lot of implementation details. One can use synthetic benchmarks during early design phases to compare different design options. Precise estimations still need to be performed when the implementation details are available.

To the best of our knowledge, this is the first work on automatically generating synthetic benchmarks from real benchmarks at the system level.

II. RELATED WORK

There are numerous studies on benchmark cloning and synthetic benchmark generation using workload characterization. Hoste et al. [3] compare micro-architecture dependent and micro-architecture independent characteristics for commonly used benchmarks. Joshi et al. [4] propose a method to analyze workloads of proprietary applications. Our work is similar to Deniz et al. [5] who generate synthetic benchmarks based on software architectural patterns for Pthreads programs and use binary instrumentation. However, they do not target system level benchmarks like us. That is, we can generate SystemC synthetics from Pthreads programs using a new system-level back-end.

Several studies estimate the performance and workload characteristics of SystemC and TLM models. SESAME [6] uses Kahn process networks for multimedia application modeling. TAPES [7] is a performance evaluation tool using transaction level simulation in SystemC. In TAPES, the applications are modeled as traces whose specification is a manual process. Similarly, Streubuhr et al. [8] propose a methodology for heterogeneous multiprocessor systems-on-chip specified at ESL level. Kreku et al. [9], [10] developed a framework which uses UML and SystemC for system-level performance evaluation. They have an abstract workload model of applications obtained using several approaches such as analytical modeling, simulation traces, and measurements. The workloads are modeled at a low level, that is, at the basic block level and contain read, write, execute instructions; whereas, we model workloads at a high level and capture the parallel pattern of the original application. They manually add counters, timers, and probes for measuring performance, whereas we obtain program behavior through automatic instrumentation and hardware counters. Their processor models have a cycles per instruction (CPI) parameter similar to us, which is used in estimating the execution time of the workloads. Grammatikakis et al. [11] developed a method to estimate the power usage of cycle and bit accurate TLM models. Number of transactions and bit transitions at each clock cycle is used to compute the relative power dissipation. Greaves et al. [12] describe a power estimation add-on to SystemC TLM modeling.

Our work differs from above mentioned SystemC/TLM works in that our goal is to generate synthetic versions of real applications with similar performance characteristics. We do not develop a system level platform model rather we develop synthetic system level applications. Our synthetics do not work at the basic block level instead we generate a synthetic based on high level parallel pattern of the application. Also, we do not depend on the source code of the application but just the binary of it. Our synthetics are regular C programs which make them portable, unlike low level assembly programs used for synthetics previously [13]. Similar to above approaches, we abstract the functionality of the application as synthetic benchmarks do not perform any useful function. None of the above SystemC/TLM works generate synthetics from real benchmarks.

III. WORKLOAD CHARACTERISTICS

In this section, we describe high-level (parallel patterns) and low-level workload characteristics used in our framework.

Patterns are high quality solutions to commonly encountered problems. Parallel software patterns are simply those that are applicable to problems related to concurrent programs. Different problems require different parallel patterns for their implementations. Mattson et al. [14] present a set of parallel patterns that are commonly used in the literature that are shown in Fig. 1.

A parallel pattern is a high level characteristic of a parallel program, hence these characteristics allow us to improve portability of our synthetic benchmarks on different platforms while preserving thread communication and data sharing behaviors as demonstrated by our experiments. Whereas, low level characteristics are dependent on the particular platform that they are obtained from. Different parallel programming patterns can be implemented in SystemC or Pthreads.

We also use low-level characteristics in determining whether the synthetic is similar to the original benchmark. The most commonly used low level characteristics for hardware performance evaluation are: Instruction-Per-Cycle (IPC), which denotes the average number of instructions per CPU cycle, Cache-Miss-Rate (CMR), which denotes the average number of cache misses per cache reference, and Branch-Misprediction-Rate (BMR), which denotes the average number of branch misses per branch instruction.

IV. SYSTEM LEVEL SYNTHETIC BENCHMARK GENERATION

We developed a synthetic benchmark generation framework that takes as input a binary program and outputs a synthetic
We presented a framework to generate system level synthetic benchmarks from multithreaded applications. To the best of our knowledge, this is the first work on generating synthetic benchmarks from real benchmarks at the system level. Our synthetic benchmarks are smaller and faster than the original benchmarks, while maintaining similar performance characteristics as the original benchmarks. Specifically, we developed system level synthetics of multicore benchmarks from the well-known PARSEC suite and obtained an average speedup of 149x. In the future, we plan to generate Pthreads synthetics from SystemC original models that will speed up the execution of SystemC models.

VI. CONCLUSIONS AND FUTURE WORK
TABLE I
PARSEC (Pthreads) ORIGINAL TO SystemC SYNTHETIC RESULTS

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ORIGINAL (Pthreads)</th>
<th>Loc</th>
<th>Parallel Pattern</th>
<th>Loc</th>
<th>Parallel Pattern</th>
<th>Speedup</th>
<th>#iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>facesim</td>
<td>20275</td>
<td>414</td>
<td>Task Parallel</td>
<td>415</td>
<td>Geometric Decomposition</td>
<td>29</td>
<td>7</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>2784</td>
<td>1095</td>
<td>Geometric Decomposition</td>
<td>7696</td>
<td>Geometric Decomposition</td>
<td>59</td>
<td>64</td>
</tr>
<tr>
<td>Blackholes</td>
<td>1262</td>
<td>223</td>
<td>Task Parallel</td>
<td>2784</td>
<td>Geometric Decomposition</td>
<td>47.5</td>
<td>13</td>
</tr>
<tr>
<td>Swaptions</td>
<td>1095</td>
<td>788</td>
<td>Task Parallel</td>
<td>788</td>
<td>Pipeline</td>
<td>230.6</td>
<td>11</td>
</tr>
<tr>
<td>Ferret</td>
<td>10/65</td>
<td>386</td>
<td>Pipeline</td>
<td>386</td>
<td>Pipeline</td>
<td>44.5</td>
<td>29</td>
</tr>
</tbody>
</table>

VII. ACKNOWLEDGEMENTS
This research was supported by Bogazici University Research Fund 7223 and the Turkish Academy of Sciences.

REFERENCES