SYSTEM LEVEL SYNTHETIC BENCHMARKS FOR MULTICORE ARCHITECTURES

by

Gökçehan Kara

B.S., Mechanical Engineering, Koc University, 2010

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2014

SYSTEM LEVEL SYNTHETIC BENCHMARKS FOR MULTICORE ARCHITECTURES

APPROVED BY:

Assoc. Prof. Alper Şen .....
(Thesis Supervisor)

Prof. Smail Niar .....

Prof. Can Özturan .....

DATE OF APPROVAL: 15.07.2014

ACKNOWLEDGEMENTS

I would like to thank my supervisor Assoc. Prof. Alper Şen for keeping me motivated, fellow jury members Prof. Smail Niar and Prof. Can Özturan for all the feedback, my mother Zühal Kara and my father Ali Sait Kara for their endless support, dear schoolmates Etem Deniz, Önder Kalacı, Ekim Boran Danışger, Çağdaş Şenol, Hasan Erdem Yantır, Görker Alp Malazgirt, Salih Bayar, Hasan Faik Alan, Yavuz Köroğlu, Rami Gökhan Kıcı and Mehmet Köse for all the brainstorming sessions we had, and the most importantly those others which I forgot to mention here for all their help.

ABSTRACT

SYSTEM LEVEL SYNTHETIC BENCHMARKS FOR MULTICORE ARCHITECTURES

System-level benchmarks enable performance evaluation early in the design cycle of multi-core architectures. However, benchmark development is costly. Synthetic benchmarks have similar performance behavior as the originals that they are generated from. Additionally, they can run faster, and they can act as proxies for proprietary customer codes that are not available in source form. In this thesis we develop a framework to generate system-level synthetic benchmarks from SystemC/TLM or Pthreads applications. These benchmarks are intended for different use cases, the former targeting virtual platforms in co-simulation environments and the latter targeting simulation platforms lacking either library support or necessary computing capabilities. Our framework was implemented by extending the synthetic benchmark generation framework developed by Deniz et al. [1] with a SystemC front-end and back-end. 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 much faster also. For example, when we generate synthetic benchmarks from the well-known multi-core benchmark suite, PARSEC, our benchmarks have an average speedup of 141x over PARSEC benchmarks. Experiments with benchmarks generated from SystemC/TLM applications also have an average speedup of 4x even for designs with the shortest execution times. We observe that our synthetics maintain similar performance characteristics as the original benchmarks and they are portable across three different multi-core architectures. Specifically, benchmarks generated from PARSEC have more than 81% similarity and benchmarks generated from SystemC have more than 88% similarly on all three architectures.

ÖZET

ÇOK ÇEKİRDEKLİ MİMARİLER İÇİN SİSTEM SEVİYESİNDE BİREŞİMSEL DENEKTAŞLARI

Sistem seviyesindeki denektaşları çok çekirdekli mimarilerde tasarım sürecinin erken safhalarında verimlilik değerlendirmesine olanak sağlamaktadır. Fakat denektaşlarının geliştirilmesinin maliyeti yüksektir. Bireşimsel denektaşları elde edildikleri asıllarıyla benzer verimlilik davranışlarına sahiptir. Buna ek olarak, daha hızlı çalışırlar ve kaynak formu mevcut olmayan tescilli müşteri kodlarının vekili olarak gorev yapabilirler. Biz bu tezde SystemC/TLM ya da Pthreads uygulamalarından sistem seviyesindeki bireşimsel denektaşları elde eden bir çatı sistemi geliştirdik. Bu denektaşları farklı kullanım senaryoları için düşünülmüş olup ilki eşbenzetim ortamlarındaki sanal platformları ve ikincisi de kütüphane desteği veya gerekli hesaplama yeteneği bulunmayan simulasyon platformlarını hedeflemektedir. Bizim çatı sistemimiz Etem ve ark. tarafından geliştirilen bireşimsel denektaşı elde etme çatı sisteminin [1] SystemC ön ve arka uçlarıyla genişletilmesiyle uygulanmıştır. Deneylerde bizim sistem seviyesindeki denektaşlarımızın elde edildikleri asıl denektaşlarına kıyasla daha küçük olmalarıyla beraber daha hızlı olduklarını gözlemledik. Mesela iyi bilinen çok çekirdekli bir denektaşı takımı olan PARSEC ile bireşimsel denektaşları elde ettiğimizde denektaşlarımız PARSEC denektaşlarına kıyasla ortalama 141 kat hızlanmaya sahip oldu. SystemC/TLM uygulamalarından elde edilen denektaşlarıyla yapılan deneyler aynı şekilde kısa çalışma süresine sahip tasarımlara rağmen ortalama 4 kat hız artışına sahip oldu. Biz bireşimsel denektaşlarımızın elde edildikleri asıllarıyla benzer verimlilik davranışları gösterdiğini ve üç farklı çok çekirdekli mimaride taşınabilir olduğunu gözlemledik. Özel olarak, üç mimaride de PARSEC ile elde edilen denektaşları %81'den fazla benzerliğe ve SystemC ile elde edilen denektaşları %88'den fazla benzerliğe sahiptir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES ix
LIST OF TABLES xii
LIST OF SYMBOLS/ABBREVIATIONS xiii
1. INTRODUCTION 1
1.1. Contributions 5
2. RELATED WORK 6
2.1. Synthetic Benchmarks 6
2.2. Performance Evaluation 7
2.3. Co-simulation in Virtual Platforms 9
3. BACKGROUND 13
3.1. SystemC 13
3.1.1. Register Transfer Level (RTL) Design 17
3.1.2. Electronic System-Level (ESL) Design 17
3.1.3. Transaction Level Modeling (TLM) 18
3.2. Parallel Patterns: High-Level Workload Characteristics 18
3.2.1. Task Parallelism (TP) Pattern 19
3.2.2. Divide and Conquer (DnC) Pattern 19
3.2.3. Geometric Decomposition (GD) Pattern 20
3.2.4. Recursive Data (RD) Pattern 20
3.2.5. Pipeline (Pl) Pattern 20
3.2.6. Event-based Coordination (EbC) Pattern 21
3.2.7. Characteristics of Parallel Patterns 21
3.2.8. Parallel Pattern Recognition 23
3.3. Low-Level Workload Characteristics 25
3.3.1. Instruction-per-cycle (IPC) 25
3.3.2. Cache-hit-rate (CHR) / Cache-miss-rate (CMR) 25

3.3.3. Branch-hit-rate (BHR) / Branch-miss-rate (BMR) 26
3.3.4. Low-Level Characteristics in Synthetic Benchmarks 27
3.4. Dynamic Binary Instrumentation (DBI) 27
4. METHODOLOGY 32
4.1. Original Benchmark Characterization 32
4.2. Original Benchmark Pattern Recognition 35
4.3. Synthetic Benchmark Synthesis 35
4.4. Synthetic Benchmark Characterization and Pattern Recognition 35
4.5. Original-Synthetic Similarity Comparison 36
5. OUR BENCHMARK GENERATION TECHNIQUE 37
5.1. Pthreads Back-end 37
5.2. SystemC Back-end 39
6. EXPERIMENTS 43
6.1. Pthreads-to-SystemC (P2S) 45
6.1.1. IPC Similarities in P2S Experiments 45
6.1.2. CHR Similarities in P2S Experiments 46
6.1.3. BHR Similarities in P2S Experiments 49
6.1.4. Overall Similarities in P2S Experiments 51
6.2. SystemC-to-Pthreads (S2P) 51
6.2.1. IPC Similarities in S2P Experiments 53
6.2.2. CHR Similarities in S2P Experiments 53
6.2.3. BHR Similarities in S2P Experiments 56
6.2.4. Overall Similarities in S2P Experiments 61
6.3. Impact of High-Level Metrics on Low-Level Metrics 61
6.4. Impact of Input Changes on Synthetics 64
6.4.1. Originals Running on Simsmall Input Set Compared to Originals Running on Simmedium Input Set 65
6.4.2. Originals Running on Simsmall Input Sets Compared to Synthetics Generated from Originals Running on Simmedium Input Sets 65
6.5. Speedups of Synthetics 69
6.6. Convergence Success Percentages of SystemC Applications 72

6.7. Iteration Counts and Times 72
7. CONCLUSIONS 75
7.1. Limitations 75
7.2. Future Work 76
APPENDIX A: PINTOOLS 77
A.1. tname_pthread.cpp 77
A.2. tname_systemc.cpp 79
A.3. shadow_pthread.cpp 81
A.4. shadow_systemc.cpp 90
APPENDIX B: P2S Example 100
B.1. blackscholes_syn.c 100
APPENDIX C: S2P Example 105
C.1. at_1_phase_syn.cpp 105
REFERENCES 121

LIST OF FIGURES

Figure 1.1. Scenarios for P2S and S2P. 3

Figure 3.1. Sample FIFO queue in SystemC. 15

Figure 3.2. (Cont.) Sample FIFO queue in SystemC. 16

Figure 3.3. Classification of parallel patterns [2]. 19

Figure 3.4. Determine the type of variables of a multi-core application [3]. . . 22

Figure 3.5. Determine the code block to be inserted. 28

Figure 3.6. Code blocks. 29

Figure 4.1. Synthetic benchmark generation work flow. 33

Figure 5.1. Pthreads back-end [1]. 38

Figure 5.2. Representative Pthreads synthetic benchmark example. 38

Figure 5.3. SystemC back-end. 39

Figure 5.4. Representative SystemC synthetic benchmark example. 40

Figure 5.5. Example SystemC synthetic. 41

Figure 5.5. (Cont.) Example SystemC synthetic. 42

Figure 6.1. IPC values in P2S experiments (individually). 47

Figure 6.2. CHR values in P2S experiments (individually). 48

Figure 6.3. BHR values in P2S experiments (individually). 50

Figure 6.4. Overall similarities in P2S experiments. 52

Figure 6.5. IPC values in S2P experiments (individually). 54

Figure 6.5. (Cont.) IPC values in S2P experiments (individually). 55

Figure 6.6. CHR values in S2P experiments (individually). 57

Figure 6.6. (Cont.) CHR values in S2P experiments (individually). 58

Figure 6.7. BHR values in S2P experiments (individually). 59

Figure 6.7. (Cont.) BHR values in S2P experiments (individually). 60

Figure 6.8. Overall similarities in S2P experiments. 62

Figure 6.8. (Cont.) Overall similarities in S2P experiments. 63

Figure 6.9. Performance characteristics of originals in input size experiments. 66

Figure 6.10. Overall similarities of originals in input size experiments. 67

Figure 6.11. Performance characteristics of synthetics in input size experiments. 68

Figure 6.12. Overall similarities of synthetics in input size experiments. 69

Figure 6.13. Iteration counts in S2P experiments. 73

Figure 6.14. Iteration times in S2P experiments. 74

LIST OF TABLES

Table 3.1. Reference behavior of data sharing characteristics [3]. 23

Table 3.2. Reference behavior of thread communication characteristics [1]. . . 24

Table 3.3. Reference behavior of general threading characteristics [1]. 24

Table 6.1. Features of PARSEC applications. 44

Table 6.2. Hardware and software specifications. 44

Table 6.3. Speedups of synthetics (P2S). 70

Table 6.4. Speedups of synthetics (S2P). 70

Table 6.5. Convergence success percentages of SystemC applications. 72

LIST OF SYMBOLS/ABBREVIATIONS

RTL	Register Transfer Level
ESL	Electronic System-Level
TLM	Transaction Level Modeling
HDL	Hardware Description Language
SoC	System-on-a-Chip
P2S	Pthreads-to-SystemC
S2P	SystemC-to-Pthreads
IPC	Instruction-per-cycle
CMR	Cache-miss-rate
CHR	Cache-hit-rate
BMR	Branch-miss-rate
BHR	Branch-hit-rate
SBI	Static Binary Instrumentation
DBI	Dynamic Binary Instrumentation
TP	Task Parallelism
DnC	Divide and Conquer
GD	Geometric Decomposition
RD	Recursive Data
Pl	Pipeline
EbC	Event-based Coordination

1. 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 thesis, we develop techniques that allow us to perform system-level performance evaluation using synthetic benchmarks.

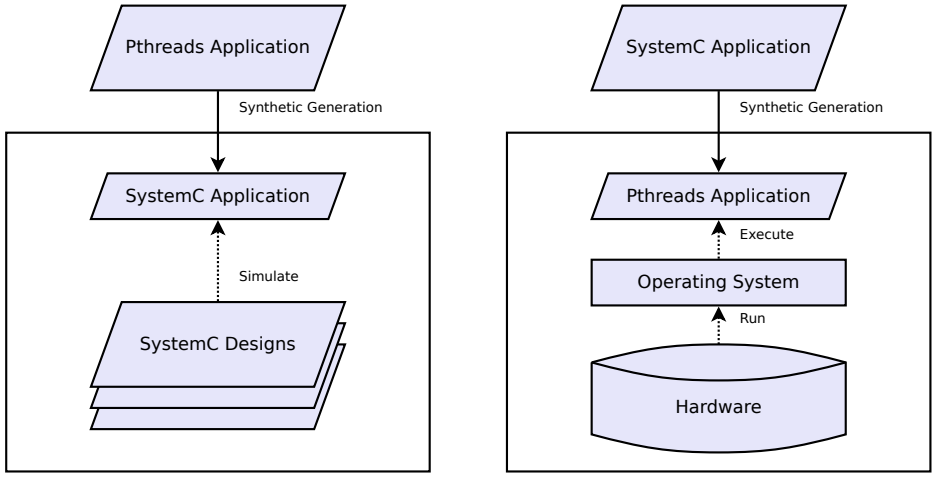
A synthetic benchmark is a small, simple and accurate program that mimics the performance characteristics of an original benchmark that it is derived from. Synthetic benchmarks have multiple advantages over traditional benchmarks in performance evaluation. First, synthetic benchmarks can be used for speeding up early architectural exploration and performance analysis. Although Transaction Level Models (TLM) [4] 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 multi-core benchmark suites such as PARSEC [5]. 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.

Our synthetic benchmark approach can be used in virtual platforms that incorporate system-level models such as SystemC [6] or TLM [4]. Virtual platforms allow early exploration of new system architectures including hardware and software components in a co-simulation environment. Co-simulation is a methodology that provides functional and/or timed simulation of mixed hardware-software systems. In this thesis, we develop techniques that can automatically generate system-level synthetic benchmarks (SystemC benchmarks) from original benchmarks that use Pthreads library (Pthreads benchmarks) or the other way around. We call these methods Pthreads-to-SystemC (P2S) and SystemC-to-Pthreads (S2P), respectively.

Technically what we do is to generate C/C++ code from C/C++ using a different library for multithreading. This is unlike other synthetic benchmark studies in the literature that generates assembly code from benchmarks at C level. Also note that we do not claim to have a perfect match for the synchronization structure found in the original in our synthetics as this is not a translator study. However, to some extend we have the same synchronization as the original as we try to capture high-level threading features in our synthetics.

The main motivation to use P2S comes from the slowdowns 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. In this scenario, we obtain a system-level synthetic version of a multithreaded application, say from PARSEC benchmark suite, that can be plugged into an existing SystemC design. This 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



(a) (b)

Figure 1.1. Scenarios for (a) Pthreads-to-SystemC (P2S) and (b) SystemC-to-Pthreads (S2P).

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. Lastly, our synthetics could be used in legacy SystemC designs where it is not possible to use applications in Pthreads. Figure 1.1a displays high level view of the mentioned scenario.

Similarly the motivation for S2P generation comes from the use of a higher performance library for synthetics. In this scenario, the original application uses the SystemC library and the synthetic application uses the Pthreads library. Pthreads library reduces some of the overhead caused by SystemC kernel during simulation. It also allows threads to run in parallel without further intervention to the SystemC implementation. The result is a speedy benchmark application that is only dependent on Pthreads

library. Figure 1.1b displays high level view of the mentioned scenario.

For synthetic benchmark generation we use various high and low-level metrics that capture the performance characteristics of applications. High-level metrics are used to describe architecture independent characteristics of the benchmarks. Among these we have metrics to measure communication and data sharing behavior of threads as well as other general threading characteristics such as creator or begin/finish times of threads. These metrics are combined to recognize the parallel pattern of the application, that is a common solution to a frequently occurring problem in software development. Low-level metrics on the other hand are used to describe performance characteristics of the benchmark on a given hardware. Most well-studied low-level performance characteristics are Instruction-per-cycle (IPC), Cache-hit-rate (CHR), and Branch-hit-rate (BHR). We aim to match low performance characteristics of the synthetics to the original on any given hardware with the aid of high-level metrics that are used in the synthesis.

For P2S generation, we perform experiments on original benchmarks from PARSEC suite. The average speedup of our SystemC synthetics is 141x compared to the original PARSEC benchmarks. All our benchmarks have more than 81% similarity to the original benchmarks on average, where the similarity is defined as the average of the error percentage between the synthetic and the original using IPC, CHR, and BHR on three different systems. For S2P generation, we perform experiments on various SystemC/TLM applications. The average speedup of our Pthreads synthetics is 4x compared to the original applications. These benchmarks have at least 88% similarity to the original benchmarks on average on all three systems. We observe that the highest speedups and similarities are obtained when the original benchmarks are large such as in the case of PARSEC benchmarks. Our experiments confirm the importance of parallel patterns in generating synthetics with high level of similarity to the original benchmark.

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.

1.1. Contributions

In summary, this study provides the following contributions.

- To the best of our knowledge, this is the first work on automatically generating synthetic benchmarks from real benchmarks at the system-level.
- We have extended the synthetic benchmark generation framework developed by Deniz et al. [1] with a SystemC front-end and back-end to enable system-level benchmark synthesis.
- Our approach speeds up system-level early architectural exploration and performance analysis through the use of smaller and faster synthetic benchmarks.
- We performed several experiments to show the preservation of performance characteristics of real benchmarks in synthetic benchmarks.

2. RELATED WORK

We inspect here various studies involving either synthetic benchmarks, performance evaluation, or co-simulation in virtual platforms. To the best of our knowledge, this is the first work on synthetic benchmark generation involving system-level SystemC and TLM programs.

2.1. Synthetic Benchmarks

Ganesan and John [7] introduces a framework to generate synthetic benchmark proxies for real world multi-threaded applications. Characterization is done by investigating data sharing patterns, stride based memory access behavior, model for memory level parallelism, transition rate based branch behavior and other dimensions of the abstract workload model. A total of 32 metrics are used for characterization. Experiments are done with the PARSEC suite. Results show an average error of 4.87% for IPC, 0.67% for L1 cache hit rate, 0.53% for branch prediction and 2.73% for power-per-cycle metric. Speedups between 4x and 6x are also possible.

Kreku et al. [8] proposes a compiler based technique for automatically generating workload models in order to use in performance simulation. Application source code is first compiled with profiling enabled in a GCC compiler extended with ABSINTH (ABStract INstruction exTraction Helper) tool. Then the compiled binary is executed with an appropriate input set in order to extract the necessary data out of it. These are used for the final compilation in the last step to generate a workload model for each function in the source code. It is also possible to map these models to a SystemC based execution platform model for simulation. Use cases are provided for a x264 video and a JPEG encoding applications in the paper.

Deniz et al. [1] introduces a framework to generate multi-threaded synthetic benchmarks based on parallel patterns for Pthreads and OpenMP programs. High-level characteristics such as data sharing, thread communication and general threading

are used for pattern recognition. These are essential in order to capture the behavior of the original application and provide portability among different machines. Low-level characteristics such as instruction-per-cycle, cache-miss-rate and branch-miss-rate are used for performance comparison. Experiments are carried out with PARSEC and RODINIA benchmarks. Results show that the average similarity score is 87% for the first hardware in which the synthesis is made and 85% for a different system. Our work extends this work with a SystemC front-end and a back-end.

Joshi et al. [9] proposes a method to analyze workloads of proprietary applications. First micro-architecture independent characteristics of the original application are extracted. The collected information is then used to generate a clone version of the application which has similar performance characteristics such as instruction mix or cache miss rate. The power of this method is that all characteristics used in the analysis are micro-architecture independent so that portability across different platforms are assured. Another advantage of this method is the reduced time of simulation without losing accuracy in performance and energy estimation. Experiments are carried out with various benchmarks including SPEC CPU2000, MiBench and MediaBench. Results show that good accuracy as well as speedups as high as 5 orders of magnitude are indeed possible.

2.2. Performance Evaluation

Coffland and Pimentel [10] introduces Sesame environment that is used for efficient design space exploration of heterogenous embedded multimedia systems. It provides simulation methods and tools used for Y-chart design approach. For this approach a designer proposes an initial architecture that would fit the requirements of target applications. Performance evaluations of different alternatives provide feedback to the designer for improvement. Sesame environment consists of application, mapping and architecture layers.

Pimentel [11] introduces Artemis workbench that builds upon Sesame envrionment to allow designers to model architectures at a higher level. The workbench uses

Compaan tool-set to convert applications written in languages such as C, C++ or Matlab to Kahn Process Network (KPN). This KPN is then given to Laura tool-set in order to produce synthesizable VHDL code out of it. Component calibration is done using Molen platform. Sesame is used as the system-level modeling and simulation environment using Y-chart design approach. A use case of Motion-JPEG encoder application is given in the paper.

Lahiri et al. [12] introduces a fast and accurate system performance-analysis technique for communication architecture designs. This technique is developed as a better alternative to simulation based and static analysis based techniques in terms of efficiency and accuracy. It is based on a hybrid trace-based performance-analysis methodology. First an abstract set of traces are extracted from an initial co-simulation in which the communication is defined through an abstract interface. Then the system designer specifies a communication architecture. Finally the corresponding performance is estimated using a Communication Analysis Graph (CAG).

SymTA/S [13] is a tool for performance and timing analysis based on formal scheduling analysis and symbolic simulation. Application model of SymTA/S consists of tasks each having one input FIFO. Each task writes its output into the input FIFO of a dependent task. Events are desribed as a set of parameters. Performance analysis method is composed of local scheduling analysis and event model propagation alternating during system-level analysis. Cyclic task dependency problem is solved using cycle analysis by considering all possibilities. SymTA/S is evaluated with a system-on-chip example in the paper.

Alameldeen and Wood [14] discusses the role of IPC on the evaluation of performance for multi-processor workloads. It is argued that the use of IPC for performance evaluation is often insufficient and even misleading. Alternative work-related metrics are proposed such as time per transaction. Results from various applications from different domains are shown in the paper.

Eyerman and Eeckhout [15] discusses performance metrics for multi-program

workloads at system-level. Two metrics are proposed in the paper, average normalized turnaround time as a user-oriented metric and system throughput as a system-oriented metric. A case study for SMT processor fetch policies are provided in the paper. Results show differences as expected in these two metrics between user and system performance for different fetch policies used.

Kunzli et al. [16] introduces a combination of simulation-based and formal methods for performance evaluation of embedded systems. Simulation based-model uses MPARM, a multi-processor cycle accurate virtual platform for SystemC. Processors are modeled in C++ using an ARM Instruction Set Simulator (ISS) called SWARM. Formal approach uses an analytical method called Real-Time Calculus (RTC). RTC helps to find worst-case bounds for memory requirements such as throughput and delay. Two converters namely S/F-Converter and F/S-Converters are implemented to build the interface between simulation and formal method. Hybrid method eliminates the need to implement everything in SystemC while also reducing the execution time compared to the simulation.

2.3. Co-simulation in Virtual Platforms

Grammatikakis et al. [17] introduces an open source SystemC-based Virtual Platform (VP). Main contribution of this study is the additional system-level monitoring layer adapted from an optimization technique originally developed for ARM Cortex-A9. This additional layer sits in between the lower system layer and the upper application layer to trace memory and CPU events in order to dynamically adjust load and memory balance. Extra overhead of the probes in this monitoring layer is quite negligible varying between 0.001% and 0.003%. As a typical use case, two applications, a concurrent matrix multiplication application and a dummy single threaded application, running on a heterogeneous multi-core system is described in the paper.

Beltrame et al. [18] introduces a novel method to model and simulate Multi-Processor System-on-Chip (MPSoC) designs using real-time and non-real-time applications. In the study, open source simulation platform ReSP is modified to have a

custom System Call emulation subsystem. This module achieves to abstract cross-compiler toolchain and Instruction Set Simulator (ISS) from the application. It also provides extensibility in HW/SW partitioning as both are modeled in the same language, SystemC, hence can be shifted towards the other one depending on the requirements. Concurrency model is implemented with Pthreads and also with OpenMP if the software is compiled with a recent GNU GCC compiler. An Earliest Deadline First (EDF) scheduling algorithm is implemented in addition to the usual two, fifo and round-robin, in order to satisfy real-time requirements. Experimental results show good performance for both real-time and computational requirements. Real-time miss-rate is below 15% and the normalized execution-time relative to execution without the real-time is above 90%.

Shen et al. [19] describes a flow for budget based design exploration for configurable processors and Multi-Processor System-on-Chip (MPSoC). Four different level for analysis from the approximate to the finest are System level, Virtual Architecture (VA) level, Transaction Accurate (TA) level and Cycle Approximate (CA) level. System Level is the highest abstraction level in which the application is described as a set of communication processes known as Kahn Process Networks (KPN). VA level includes inter-subsystem communication to system level analysis. TA level is used to model approximate timing information of the subsystem without giving up simulation speed. Finally in CA level, all hardware components are modeled using cycle-approximate TLM to find the most approximate timing results using an Instruction Set Simulator (ISS). A Motion-JPEG case study is given in the paper beginning from exploration in TA level to CA level to achieve speedups as high as 131 times.

Chureau and Pétrot [20] introduces a design flow for Multi-Processor System-on-Chips (MPSoC) that is agnostic to different abstraction levels. An interfacing method called Service-Based Intermediate Format (Serif) is proposed as an intermediate format that would allow HW and SW to co-simulate at all levels. Serif consist of tasks that are defined by an interface bundled with a behavior and the required data. Interfaces themselves can be of two types, either a communication module to exchange data between tasks or a logistic module which maps software tasks. This format is

compatible with existing SystemC components and these can be imported with the importation tool called SystemC2Serif. Experimentations are done over an MJPEG decoder application at system and transaction accurate levels.

Posadas et al. [21] explores the use of SystemC for Real-Time Operating Systems (RTOS). SystemC already provides primitives to allow low-level timing accurate simulation, however RT/E SW requires some additional RTOS concurrency and synchronization features such as time-slicing and priority based preemption. This study proposes a time manager module implemented on top of SystemC kernel in order to estimate the time for SW execution. POSIX standard is provided as the API with the addition of 180 custom functions and mapping the rest to the underlying OS functions. A complex case study of the standard EN 301 245 vocoder is given in the paper. Overhead of the additional POSIX API over the normal simulation kernel is measured to be 6HW/SW communication during co-simulation is not discussed in the study.

Fummi et al. [22] introduces a semi-homogeneous simulation environment that is using SystemC to model HW and C/C++ as the SW. Two execution environments, board-based and ISS-based (Instruction Set Simulator), are compared for the SW execution. The main characteristics of board-based approach is that it can abstract the HW component from the simulation while requiring modifications for the SW component. Conversely, ISS-based approach requires knowledge about the underlying HW component although it can run arbitrary SW component without modification. ISS-based approach promises based on three concepts, ISS-centric co-simulation, abstraction of HW parts surrounding the CPU and modularity of HW parts. On the other hand, in board-based approach communication between HW and SW is done using a driver-based scheme and the timing synchronization is done with virtual ticks. Both approaches require modification of the SystemC kernel.

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 but 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 [7]. 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.

3. BACKGROUND

In this section we give a brief description of SystemC language and some of its specific features related to our study. We also provide information about some common design approaches used in industry as we have such examples in our experiments. We then describe parallel patterns and methods used to recognize these patterns. We use parallel patterns to provide portability in our synthetics by capturing high-level characteristics of original applications. After that we mention micro-architecture dependent characteristics used in performance studies. Synthetic benchmarks are expected to match these performance characteristics with their originals. Lastly we describe dynamic binary instrumentation method which we use for benchmark characterization.

3.1. SystemC

SystemC [6] is a language which is used to model systems having both hardware and software components. It is a library built on top of C++ language rather than a language on its own like other hardware description languages (e.g. Verilog or VHDL). The standard and a reference implementation is currently maintained by Accellera Systems Initiative as an open source project.

Since hardware models are inherently parallel, SystemC provides fundamental multi-threaded design components such as threads and processes as built-in structures. These are mapped to native operating system threads (i.e. Pthreads or Qthreads) during the simulation without giving up the control of scheduling. Instead SystemC kernel uses a non-preemptive event-driven method which executes threads and processes according to the occurrence of events. This is accomplished by registering processes sensitive to particular events. As SystemC programs have multiple threads, it is possible to parallelize the simulation kernel [23, 24].

We provide here a sample implementation of a FIFO queue in SystemC in order to better explain the commonly used constructs of the language. This is a slightly mod-

ified version of `simple_fifo` example that comes with the Accellera Systems Initiative SystemC/TLM distribution [25]. Implementations of methods are mostly regular C++ code and not provided here not to make any clutter.

SystemC provides building blocks such as ports/exports, interfaces and channels for communication. Figures 3.1a and 3.1b are interface classes for `sc_interface` class while Figure 3.1c shows an implementation class extending `sc_channel` and the rest.

Modules are the basic building blocks in a SystemC design. Modules can contain one or more `SC_THREAD`, `SC_METHOD` or `SC_PROCESS`. These are just C++ macros provided for threading to resemble a traditional class hierarchy as seen in other Hardware Description Languages (HDL) along with others macros such as `SC_MODULE` and `SC_CTOR`. Figures 3.2a and 3.2b provides a producer and a consumer class each having simply an input or an output port as well as a thread for processing. These are wrapped up and connected using the fifo class within a top module in Figure 3.2c and then simply instantiated in the main function in Figure 3.2d.

In addition to these SystemC also provides special purpose data types such as `sc_int` or `sc_bit`. One can use either these types or built-in C++ data types in the design. The difference between the two is that the former is more customizable while the latter is usually faster.

Complexity of today's systems require a high-level approach to the design and verification of electronic systems. Electronic System-Level (ESL) design was born out of these needs and is currently widely accepted by the industry while Transaction Level Modeling (TLM) is a set of requirements to introduce ESL standards to the SystemC language as well as other system languages. As of this writing, TLM library has been merged into the standard SystemC package (version 2.3.0).

```
class write_if
  : virtual public sc_interface {
  public:
    virtual void write(char) = 0;
    virtual void reset() = 0;
};
```

(a)

```
class read_if
  : virtual public sc_interface {
  public:
    virtual void read(char &) = 0;
    virtual int num_available() = 0;
};
```

(b)

```
class fifo
  : public sc_channel,
    public write_if,
    public read_if {
  public:
    fifo(sc_module_name name)
      : sc_channel(name),
        num_elements(0),
        first(0) {}

    void write(char c) { .. }
    void read(char &c) { .. }
    void reset() { .. }
    int num_available() { .. }

  private:
    enum e { max = 10 };
    char data[max];
    int num_elements, first;
    sc_event write_event, read_event;
};
```

(c)

Figure 3.1. Sample FIFO queue in SystemC showing (a) a write interface (b) a read interface and (c) a fifo channel.

```
SC_MODULE(producer) {
  public:
    sc_port<write_if> out;

    SC_CTOR(producer) {
      SC_THREAD(main);
    }

    void main() { .. }
};
```

(a)

```
SC_MODULE(consumer) {
  public:
    sc_port<read_if> in;

    SC_CTOR(consumer) {
      SC_THREAD(main);
    }

    void main() { .. }
};
```

(b)

```
SC_MODULE(top) {
  public:
    fifo *fifo_inst;
    producer *prod_inst;
    consumer *cons_inst;

    SC_CTOR(top) {
      fifo_inst = new fifo("Fifo1");

      prod_inst = new producer("Producer1");
      prod_inst->out(*fifo_inst);

      cons_inst = new consumer("Consumer1");
      cons_inst->in(*fifo_inst);
    }
};
```

(c)

```
int sc_main (int, char *[]) {
  top top1("Top1");
  sc_start();
  return 0;
}
```

(d)

Figure 3.2. (Cont.) Sample FIFO queue in SystemC showing (a) a producer (b) a consumer and (c) a top module as well as (d) the main function.

3.1.1. Register Transfer Level (RTL) Design

Register Transfer Level is the commonly used abstraction for system design in industry today. It allows design to be developed using hardware registers, signals between hardware registers and other basic logical operations. VHDL and Verilog are the most commonly used RTL design languages in use today.

RTL design can then be converted directly to synthesizable gate level description by a logic synthesis tool. This is conceptually similar to compilation in software programs. Optimizations are also done at this stage.

3.1.2. Electronic System-Level (ESL) Design

Nowadays Integrated Circuits (IC) consist of millions of gates on a single chip, hence the only way to deal with the complexity is to use an abstraction. Register Transfer Level (RTL) design approach is commonly used these days within Hardware Description Languages (HDL) such as Verilog and VHDL which allows systems to be expressed with signals between hardware registers and logical operations modifying these signals. Even with RTL abstraction, it becomes impossible to design System-on-Chip (SoC) systems or Field-Programmable Gate Arrays (FPGA) which requires millions of lines of RTL code.

Electronic System-Level (ESL) abstraction is an emerging approach to deal with today's design complexity. Higher level languages such as C, C++, MATLAB or other graphical modeling tools such as Simulink are used to design systems consisting of hardware, software and other algorithmic requirements.

Also with the increasing complexity, it became impossible for a designer to know everything about the system. As a consequence, today's systems are modeled at different stages such as hardware design, hardware verification and software development. This approach requires an intermediate language to represent a system architectural model which is aimed to provide a bridge between different stages. Use of a higher level

language in ESL provides the necessary communication between these different stages in the development whereas waterfall as a traditional solution seems to lack efficiency because most systems today need concurrent development by different teams.

3.1.3. Transaction Level Modeling (TLM)

Transaction Level Modeling library provides a use for ESL design in SystemC language. TLM works as an Application Programming Interface (API) for block level model communication. A number of interface classes exist to provide channels which is an abstraction for the buses, FIFO's and other underlying details as well as to give opportunity for interoperability between different languages.

According to TLM 2.0 standard, transactions are made between an `Initiator` and a `Target` structure. All communications are made using a `generic payload` object. These structures are implemented as interface classes in `tlm` namespace. Simple implementations extending these interfaces are also available in the current TLM 2.0 distribution under `tlm_utils` namespace.

3.2. Parallel Patterns: High-Level Workload Characteristics

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. A parallel pattern language is defined in [2] and we show algorithmic patterns of this language in Figure 3.3.

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 will be demonstrated by 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 will now describe these

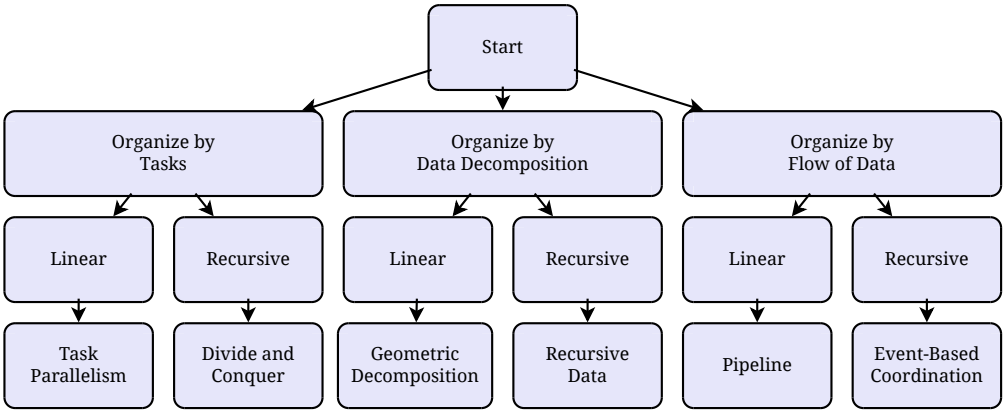


Figure 3.3. Classification of parallel patterns [2].

patterns in more detail together with the communication and data sharing behavior in the patterns.

3.2.1. Task Parallelism (TP) Pattern

Task parallelism pattern is used when there are tasks that can be executed concurrently. Typically tasks differ from each other and there is no need for communication between threads. Examples of problems that are applicable for task parallelism pattern include ray tracing (path of each ray can be computed independently from each other), molecular-dynamics problems (different type of forces on molecules are sometimes independent of each other and can be calculated as different tasks) and branch and bound problems (such as checking for the existence of an item in a collection where solution space can be divided into multiple sections and searched concurrently without waiting for each other). This pattern is most common when different SystemC modules execute different tasks in parallel with little or no communication.

3.2.2. Divide and Conquer (DnC) Pattern

Divide and Conquer pattern is suitable for recursive algorithms where the problem can be broken down to two or more sub-problems of the same type. The most typical

examples of this pattern are merge sort (where the original array is divided into multiple sub-arrays and then sorted ones are merged together) and quicksort (where the division is done using an arbitrary pivot element so that sorted sub-arrays are strictly greater or smaller than each other when being merged). We expect to have none to few thread communication in this pattern and more of shared variable types rather than private variables, which we will describe later.

3.2.3. Geometric Decomposition (GD) Pattern

This pattern is used when the data can be decomposed into chunks that can be updated concurrently. Many matrix operations (matrix multiplication, mesh computations for numerical heat transfer problems) can be computed concurrently using this pattern. Threads send data to each other at each step whenever there is a need for it, hence we expect to have few to many thread communication depending on the problem and also many shared variables that are mostly read but not written.

3.2.4. Recursive Data (RD) Pattern

Recursive problems which have dependencies like a graph or a tree fall into this category. This pattern is similar to divide and conquer pattern with more irregular, badly-defined dependencies. We also expect to have more private and read-only variables than others. Examples include finding the root of a directed tree or prefix summation of an array. This pattern is rarely used in SystemC.

3.2.5. Pipeline (Pl) Pattern

Pipeline is a common concept having examples in different domains like instruction pipelines or shell pipes. Some problems are best computed using multiple threads for different stages of the computation. Examples include signal or image processing where multiple filters can be applied one after another by different threads or Fourier-transform computations. Characteristics of this pattern is few thread communication and shared variables that migrate between threads. TLM models often use this pattern

when passing data between different modules.

3.2.6. Event-based Coordination (EbC) Pattern

Event based coordination pattern is used whenever there are tasks that are semi-independent but the ordering that is defined by the data flow is not linear as in pipeline. Some of the tasks can be executed in parallel while others need to wait for each other and even in some cases there could be loops among tasks. This pattern is rarely used in SystemC.

3.2.7. Characteristics of Parallel Patterns

Poovey et al. [3] uses data sharing characteristics for parallel pattern recognition. Four variables are defined for this purpose, namely read-only, migratory, producer/consumer and private variables. Variable types are determined according to the numbers of readers and writers at that memory location as described in Figure 3.4.

Deniz et al. [1] improves upon this study by adding different metrics. Thread communication is categorized under three classes:

None refers to terminal thread or minimal communication.
Few refers to thread communication defined algorithmically.
Many refers to data dependent case.

Other characteristics are examined under the general threading category. These characteristics are as follows:

PC (unique) is the uniqueness of program counters among threads. It is equivalent to the address of the beginning function of a thread. This is used to determine whether threads are executing the same function or not.

Thread balanced is how balanced the workloads of threads are to each other. Workload of threads are measured using the count of executed instructions. Depending

r : *tid of the last reader*

w : *tid of the last writer*

nr : *number or readers*

nw : *number or writers*

Require: $r,\ w,\ nr,\ nw$ *are counted for the corresponding variable*

Ensure: *variable type is determined accordingly*

if $nr = 1\ \&\ nw = 1\ \&\ r = w$ **then**
    private variable
else if $nw = 0$ **then**
    read only variable
else if $nr > nw$ **then**
    producer consumer variable
else
    migratory variable
end if

Figure 3.4. Determine the type of variables of a multi-core application [3].

Table 3.1. Reference behavior of data sharing characteristics [3].

	TP	**DnC**	**GD**	**RD**	**Pl**	**EbC**
Read-only	***		*	***		
Migratory	**	***	**		****	****
Prod/Cons		**	****			
Private	****	*	*	****	*	*

on the pattern of the application, threads can be balanced or unbalanced.

Creator (same) is whether or not threads are created by the same thread or not. This is used to find the fork/join behavior of the application. Different patterns have different characteristic fork/join behaviors.

Lifetime (same) is how much the lifetimes of threads are similar. It is calculated simply by differentiating creation and exit times of threads. Note that having similar lifetimes does not necessarily imply that threads are created or exited at the same time.

Creation times is how close thread creation times are to each other. Some of the patterns require creating all threads in the beginning while others need to delay the creation.

Exit times is how close thread exit times are to each other. In some patterns threads finish execution at the same time while in others some of them may finish earlier.

3.2.8. Parallel Pattern Recognition

Characteristics given in the previous section are used to recognize the parallel pattern of a program in corresponding studies. Reference behaviors of different parallel patterns are used to find the closest match of a given program. Poovey et al. [3] only uses data sharing characteristics for this purpose. Weights of reference behaviors used in this study are provided in Table 3.1.

Deniz et al. [1] improves upon this study by adding thread communication and

Table 3.2. Reference behavior of thread communication characteristics [1].

	TP	**DnC**	**GD**	**RD**	**Pl**	**EbC**
None	****	***			*	
Few	*	**	*	*	****	****
Many		*	****	****		

Table 3.3. Reference behavior of general threading characteristics [1].

	TP	**DnC**	**GD**	**RD**	**Pl**	**EbC**
PC (unique)	***	**	****	****		
Thread balanced	**	*	****	**	**	***
Creator (same)	****	**	****	***	***	*
Lifetime (same)	**	**	****	*	***	**
Creation times	***	*	****	**	***	**
Exit times	**		***		**	*

general threading metrics and their corresponding reference behaviors. We show the weights of these other reference behaviors in Tables 3.2 and 3.3, respectively. They measure the distance between the reference behavior and the results from the characterizer to recognize the parallel pattern, which is simply finding the solution to the k-nearest neighbors algorithm with Euclidean distance, where k is 1. The reference pattern closest to the results is the pattern of the benchmark.

For example, the parallel pattern of a benchmark with read-only and private data sharing characteristics, no inter-thread communication, unique thread PCs that are created by the same thread is task parallel. Similarly, an example of geometric decomposition pattern can be a benchmark with many producer/consumer and few migratory data sharing characteristics, many data dependent inter-thread communication and balanced threads that share the same PC and created by the main thread

at the same time. Experimental works show that they have a very high accuracy in recognizing the correct pattern.

3.3. Low-Level Workload Characteristics

In determining whether the synthetic is similar to the original benchmark, we check whether the synthetic has the same performance characteristics as the original. For this purpose, we use low-level (micro-architecture dependent) characteristics. Typically these characteristics differ from hardware to hardware. Most hardware comes with performance counters to obtain these characteristics. We use Linux perf tool [26] to read these counters. Brief descriptions of the performance characteristics we use are given below.

3.3.1. Instruction-per-cycle (IPC)

This is the average number of instruction per a CPU cycle and typically a number between 0 and 1 for a single core as most instructions require more than one cycle to complete. A value that is close to 1 may be an indication for Reduced Instruction Set Computer (RISC) architectures where instructions are usually simple and most of them takes only one CPU cycle whereas a smaller value may indicate Complex Instruction Set Computer (CISC) architectures where one can find quite complex instructions in the instruction set. In order to calculate this value one can simply use the formula:

$$IPC = \frac{total\ number\ of\ instructions}{total\ number\ of\ cycles}$$

3.3.2. Cache-hit-rate (CHR) / Cache-miss-rate (CMR)

Caches are fast memories that are used by CPU's in order to reduce the average memory access time by storing the most frequently used data. They are smaller in size compared to the main memory as there is a trade-off between memory speed and price. In modern CPU's there are usually different levels of caches (e.g. L1 cache, L2 cache) starting from the fastest and smallest to slowest and biggest.

The term *cache-hit* refers to an event where the required data is found on the cache hence main memory access is avoided. Similarly *cache-miss* refers to the opposite case where an item is not available on the cache yet and therefore brought to the cache from main memory. A common formula for CHR is:

$$CHR = \frac{total\ number\ of\ cache\ hits}{total\ number\ of\ cache\ references}$$

CMR can be calculated by either subtracting CHR from 1 or by using the formula:

$$CMR = \frac{total\ number\ of\ cache\ misses}{total\ number\ of\ cache\ references}$$

Since it's not possible to fit all the data into the cache, there are different algorithms to decide which elements to replace when a new one comes. This in conjunction with the cache-size is the main factor effecting the cache-miss/hit rates. For high performance we want to have cache-miss rate to be close to 0 and at the same time cache-hit rate to be as high as possible that is close to 1.

3.3.3. Branch-hit-rate (BHR) / Branch-miss-rate (BMR)

Modern CPU's use an instruction pipeline in general, that is, they start executing future instructions before the current one finishes. Whenever there is a branch in the code introduced by a conditional, it's not possible to guess the execution path beforehand. Instead of waiting for the current instruction to finish, CPU's usually start executing the instructions in one of the branches and revert back to the beginning of the other branch in case the branch taken turns out to be the false one. A correct branch prediction is referred to as a *branch-hit*, whereas an incorrect prediction is referred to as a *branch-miss*. For high performance, we want to have a branch-miss rate that is close to 0. The calculation of BHR is done according to the formula:

$$BHR = \frac{total\ number\ of\ branch\ hits}{total\ branch\ instructions}$$

BMR can be calculated by either subtracting BHR from 1 or by using the formula:

$$BMR = \frac{total\ number\ of\ branch\ misses}{total\ branch\ instructions}$$

Most CPU's take a branch deterministically instead of choosing at random. There are various techniques to increase the accuracy of predictions. State-of-the-art technology can achieve branch hit rates as high as 90%.

3.3.4. Low-Level Characteristics in Synthetic Benchmarks

Deniz et al. [1] introduces an evolutionary algorithm to match low-level characteristics in synthetic benchmarks. The code blocks for low-level metrics are added to the main function of the candidate synthetic benchmark. For each low-level metric we define an error percentage as follows:

$$Error = \frac{|Original - Synthetic|}{Original}$$

Similarity then simply becomes:

$$Similarity = 1 - Error$$

At each iteration step we pick the metric with the lowest similarity score and try to make it closer to the original which is depicted in Figure 3.5. For IPC values we use integer addition for increment and integer division for decrement. For CMR values we try to access data that is not cached for increment and vica versa. BMR values are handled in a similar fashion. In Figure 3.6 we show the code blocks used in the synthesis for these metrics.

3.4. Dynamic Binary Instrumentation (DBI)

Instrumentation is the process of putting additional instructions to specific portions of the program in order to collect information about the execution. This information could be used for different purposes such as debugging, logging or as in our case

Require: *IPC, CMR, or BMR having the lowest similarity score is determined*

Ensure: *the most appropriate code block is inserted*

if *lowest is IPC* **then**
  if $IPC_{org} < IPC_{syn}$ **then**
    insert IPC increment block
  else
    insert IPC decrement block
  end if
else if *lowest is CMR* **then**
  if $CMR_{org} < CMR_{syn}$ **then**
    insert CMR increment block
  else
    insert CMR decrement block
  end if
else if *lowest is BMR* **then**
  if $BMR_{org} < BMR_{syn}$ **then**
    insert BMR increment block
  else
    insert BMR decrement block
  end if
end if

Figure 3.5. Determine the code block to be inserted.

```
// IPC increment block
int i1 = 288, i2 = 3, ires = 0;
for (workCount = 0;
     workCount < /* .. */;
     workCount++) {
  ires = i1 + i2;
  ires = i1 + i2;
  // ..
  // 40 times
  // ..
  ires = i1 + i2;
}
```

(a)

```
// IPC decrement block
double d1 = 288, d2 = 3, dres = 0;
for (workCount = 0;
     workCount < /* .. */;
     workCount++) {
  dres = d1 / d2;
  dres = d1 / d2;
  // ..
  // 40 times
  // ..
  dres = d1 / d2;
}
```

(b)

```
// CMR increment block
register int wi;
int *wa, wsize = (1024 * 1024) * 4;
wa = (int *)
  malloc(wsize * sizeof(int));
for (workCount = 0;
     workCount < /* .. */;
     workCount++) {
  wa[rand() % wsize] = 0;
}
free(wa);
```

(c)

```
// CMR decrement block
int wa[256][256];
for (workCount = 0;
     workCount < /* .. */;
     workCount++) {
  for (i = 0; i < 256; i += 1) {
    for (j = 0; j < 256; j += 1) {
      wa[i][j] = 2 * wa[i][j];
    }
  }
}
```

(d)

```
// BMR increment block
int b1 = 288, b2 = 3, bres = 0;
unsigned int randNum, r3, r4, r8;
for (workCount = 0;
     workCount < /* .. */;
     workCount++) {
  randNum = rand();
  r3 = randNum % 3;
  if (r3 == 0) { bres = b1 + b2; }
  if (r3 == 1) { bres = b1 + b2; }
  r4 = randNum % 4; // ..
}
```

(e)

```
// BMR decrement block
int b1 = 288,
    b2 = 3,
    bres = 0;
for (workCount = 0;
     workCount < /* .. */;
     workCount++) {
  if (workCount >= 0) {
    bres = b1 + b2 + (b1 / b2);
    bres = b1 + b2 + (b1 / b2);
  }
}
```

(f)

Figure 3.6. Code blocks for (a) IPC increment, (b) IPC decrement, (c) CMR increment, (d) CMR decrement, (e) BMR increment and (f) BMR decrement.

profiling. Two basic types of instrumentation are source code instrumentation and binary instrumentation. We use binary instrumentation for its advantage of avoiding the need to access the source code of programs as this is usually the main reason driving people into using synthetic benchmarks.

Instrumentation can be done at different granularities. Some of these are as follows:

Instruction An instruction is a single operation a processor can handle. Different architecture have different instruction sets, some of them being more complex than others (i.e. RISC vs CISC) and may contain multiple memory references within.

Basic block (BBL) Basic blocks form nodes and vertices in a control flow graph. They have one entry and one exit point each hence no jump instruction. These are commonly used for analysis in compilers.

Trace Like basic blocks, a trace has a single entry point but may contain multiple exit points, therefore a trace usually consists of multiple basic blocks. This level is mostly useful for optimizations in binary instrumentation frameworks.

Function A function is a subroutine in an execution that is responsible for a specific task. In object oriented languages, a function that belongs to a class is referred as methods. Functions and methods are very similar for binary instrumentation frameworks.

Binary instrumentation can further be classified as Static Binary Instrumentation (SBI) where the executable is changed before the execution starts and Dynamic Binary Instrumentation (DBI) where the executable is changed on the fly (i.e. instructions are interpreted by the binary instrumentation framework with possible additions wherever necessary). The key difference is that because of the fact that DBI happens at run time, it is easier to instrument dynamic library routines. We use Pin tool [27] for DBI. It is developed by Intel and is available in Linux and Windows for both 32 and 64 bit architectures. There are also other binary instrumentation frameworks such as Valgrind [28] and DynamoRIO [29].

Two instrumentation modes are supported by Pin, Just-in-Time (JIT) mode and Probe mode. JIT mode creates a modified copy of the application on-the-fly and is more commonly used compared to probe mode. Probe mode on the other hand, provides a higher performance with a lower overhead in trade of losing some flexibility. In probe mode, program is executed on the original code with some additional jump instructions for probing. Some of the Pin API is not available in Probe mode. We use JIT mode in our framework.

In our framework, we use dynamic binary instrumentation to collect high-level characteristics from the execution of Pthreads or SystemC applications. Source code of our instrumentation clients in Pin framework can be found in Appendix A.

4. METHODOLOGY

We have extended synthetic benchmark generation framework developed by Deniz et al. [1] with a SystemC front-end and back-end. The end result takes an input as either Pthreads or SystemC binary and outputs a synthetic program that mimics the behavior of the input program using the other library. Figure 4.1 shows the steps involved in this synthesis process. Our work flow begins with collecting information from the original application using both Linux perf tool [26] and our clients in Pin DBI framework. Afterwards we use this information to generate an initial version of the synthetic program with the same recognized parallel pattern as the original one. We then instrument the synthetic program so that we can converge to similar performance values of the original program by iteratively adding necessary code blocks to the synthetic program. This evolutionary algorithm continues until the user specified similarity thresholds are satisfied. Details of each step are given in following sections.

4.1. Original Benchmark Characterization

We obtain high-level and low-level characteristics of the original at this step. Low-level characteristics (IPC, CMR, and BMR) are measured using Linux perf tool. These are frequently used in performance evaluations [30, 31]. High-level characteristics are used as micro-architecture independent characteristics [32]. To obtain high-level characteristics, we implemented instrumentation clients in Pin framework given in Appendix A. Our Pintool clients measure data sharing characteristics such as the number of different types of variables (private, read-only, producer-consumer, migratory) and thread communication behaviors (dependency between threads and communication among threads). Our client also measures other general threading behaviors such as the program counter of functions executed by threads (to determine whether threads are executing the same task or not), the creator of threads, instruction counts of threads (to determine whether threads are balanced or unbalanced), the number of load/store instructions, and the lifetimes of threads.

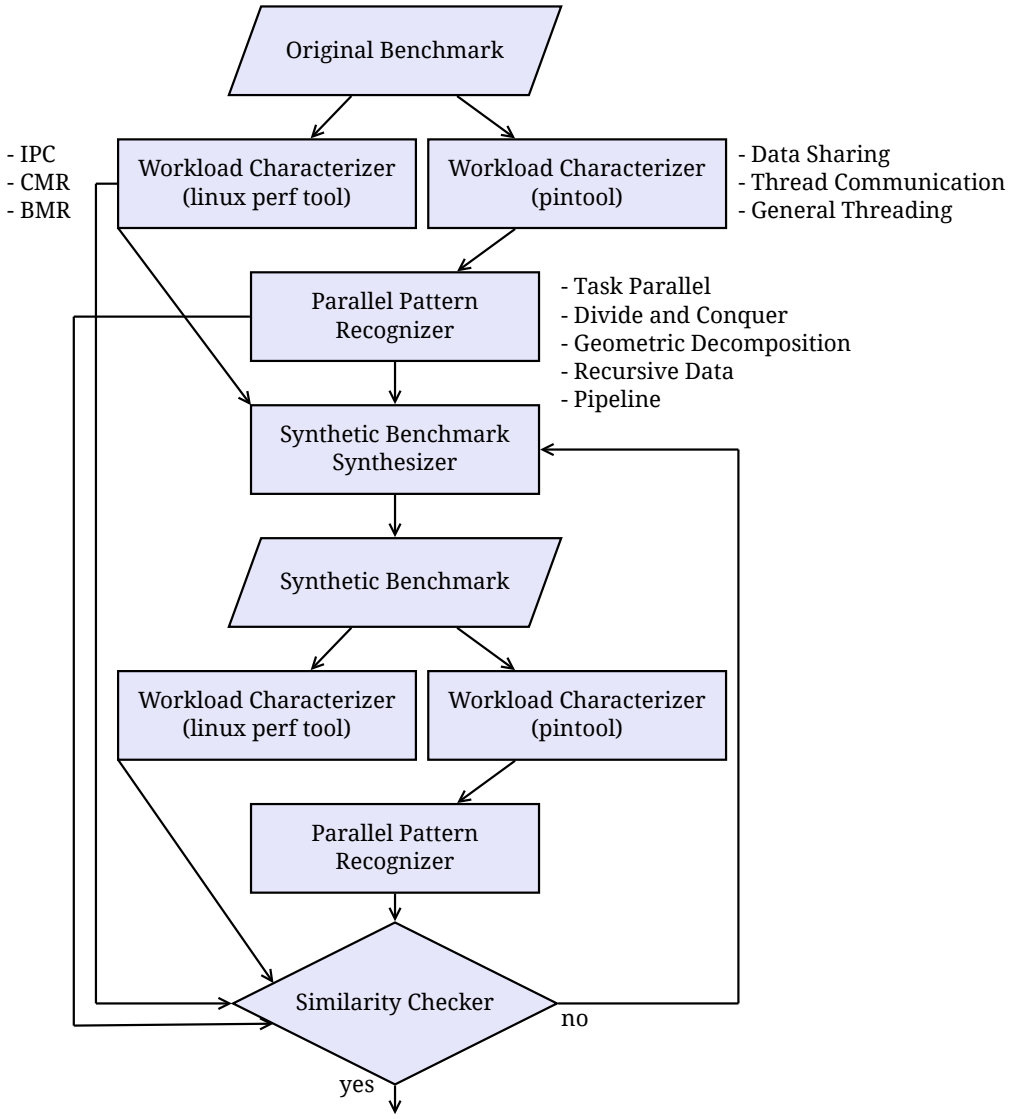


Figure 4.1. Synthetic benchmark generation work flow.

We can determine the functions executed by threads by instrumenting calls to `pthread_create` in Pthreads or `create_thread_process` in SystemC libraries and then reading the appropriate arguments to these calls. Extracting thread function names are done in a separate run because SystemC kernel registers threads at runtime and Pin does not allow setting additional callbacks after the program starts. This has not become an issue since SystemC threads are static and do not change from run to run and we collect all performance characteristics in a single separate second run using a different client. SystemC kernel creates all threads at initialization step, executes them according to its own schedule and then finally destroys them all together just before the simulation ends. Hence, we cannot use the built-in thread start and finish events provided by Pin framework. In order to accurately measure lifetimes of threads we extract SystemC thread names as described above. Then we record the start and finish times of these thread functions. This approach is also necessary not to get the same program counter for all threads pointing to the same SystemC thread creation routine.

To extract the type of variables, we have implemented a simple shadow memory algorithm in Pin that is updated with read/write instructions on memory locations. Our shadow memory algorithm does not model cachelines but rather uses each memory reference, hence false sharing that results from addressing multiple addresses in the same cacheline is prevented. Another advantage of our implementation is that we have parametrized the maximum number of threads using a C++ `bitset` rather than relying on the number of bits of built-in primitive types such as `int` and `long` by sacrificing some performance/memory space. However we believe this is the right approach since typical number of threads in a hardware design may easily exceed any fixed number like 32.

After the execution is complete, we check the shadow memory for each memory location to count the number of reader/writer threads for that location. Using these counts, we are able to determine the variable types in the execution according to Figure 3.4.

4.2. Original Benchmark Pattern Recognition

As the pattern is one of the most important characteristics of a parallel program, we use results of our Pintool client to correctly recognize the parallel pattern. We use the pattern recognizer of the framework developed by Deniz et al. [1] using the method described before in Subsection 3.2.8.

4.3. Synthetic Benchmark Synthesis

Using the parallel pattern of the original benchmark from the previous step, we generate an initial version of the synthetic. This initial version has code blocks for individual threads such that they have the same high-level characteristics of the original but does not yet contain any code blocks to match low-level characteristics. For a matrix multiplication application, which has geometric decomposition pattern, we create as many threads in the synthetic as there are in the original such that these threads execute the same function. Then, we create shared variables and read/write operations on them to mimic many data dependent inter-thread communication characteristics.

4.4. Synthetic Benchmark Characterization and Pattern Recognition

The characterization and pattern recognition of the synthetic benchmark is similar to the steps for the original benchmark described above. If the initial synthetic does not have the same parallel pattern as the original, then we update the synthetic by addition of high-level characteristics code blocks to the individual threads until we match the patterns. This is because having a correct parallel pattern is the most important factor in generating high quality synthetics. In other words, if one only preserves low-level metrics in the synthetic the synthetic is not going to be similar to the original.

4.5. Original-Synthetic Similarity Comparison

In the last step, we have high and low-level characteristics for both the original and synthetic programs. Furthermore, the synthetic and the original have the same parallel pattern. We now check to see the amount of similarity between the synthetic application and the original with respect to the low-level metrics. The user sets the both overall similarity score and the individual similarity scores. The individual similarity scores are given for IPC, CMR, and BMR and the overall similarity score is the average of IPC, CMR, and BMR individual scores. If the similarity score is not met, we loop back to the synthetic benchmark synthesis step to generate a new synthetic by adding code blocks for low-level characteristics. In the new generation step, we pick the most dissimilar characteristic and try to match it by adding/removing code blocks such as those shown in Figure 5.5 for IPC, CMR, and BMR.

5. OUR BENCHMARK GENERATION TECHNIQUE

In this chapter we provide comparative information about the existing Pthreads back-end [1] and our new SystemC back-end implementations. In order to better explain our work, we provide simple examples of SystemC synthetic programs used in our experiments. Full sized synthetic benchmark examples can be found in Appendix B for P2S generation and Appendix C for S2P generation. Priority is given to the SystemC back-end as it is the main contribution of this study. Pthreads synthetics have the same kind of code blocks and threading structure except for module and class constructs.

We skip some of the details regarding the generation of boilerplate code such as inclusion of header files or generating definitions and structures. The existing Pthreads backend is modified slightly to match the new SystemC back-end accordingly for these tasks. We also adapted the existing code for generating the content of each thread and the main thread for the SystemC back-end. Low-level and high-level characteristics are infused at this stage. Therefore the main difference between Pthreads and SystemC back-ends is the threading structure. These are given in the following sections.

5.1. Pthreads Back-end

Pthreads back-end is used for S2P generation. In S2P generation the original SystemC application is analysed with our SystemC front-end and then the result is given to the existing Pthreads back-end [1] for synthesis. Pthreads back-end creates as many threads as the original SystemC design has. It then creates and destroys these threads in the main function using `pthread_create` and `pthread_join`. Algorithm for Pthreads back-end is given in Figure 5.1. An example output showing the threading structure is provided in Figure 5.2.

Require: *data for threads are prepared for synthesis*

Ensure: *the correct synthetic is generated*

for all *threads* **do**

generate thread signature

generate thread content

end for

generate main thread signature

generate main thread content

Figure 5.1. Pthreads back-end [1].

```
..
void *task0(void *param) { .. }
void *task1(void *param) { .. }
..

int main(int argc, char **argv) {
  ..
  pthread_create(&threads[0], NULL, task0, (void *) &tData[t]);
  pthread_create(&threads[1], NULL, task1, (void *) &tData[t]);
  ..
  for (t = 0; t < number_of_threads; t++) {
    pthread_join(threads[t], NULL);
  }
  ..
}
```

Figure 5.2. Representative Pthreads synthetic benchmark example.

Require: *data for threads are prepared for synthesis*
Ensure: *the correct synthetic is generated*

generate module signature
for all *threads* **do**
    generate thread signature
    generate thread content
end for
generate constructor signature
for all *threads* **do**
    register thread
end for
generate main thread signature
generate main thread content

Figure 5.3. SystemC back-end.

5.2. SystemC Back-end

SystemC back-end is used for P2S generation. In S2P generation the original Pthreads application is analysed with our Pthreads front-end and then the result is given to our SystemC back-end for synthesis. In SystemC back-end threads are wrapped in a dummy SystemC module and registered in the module constructor. Algorithm for SystemC back-end is given in Figure 5.3. An example output showing the threading structure is provided in Figure 5.4.

In Figure 5.5, we show the SystemC synthetic generated using our framework for the task-parallel PARSEC Blackscholes application. Note that we compile the synthetic with -O0 flag, that is, with no optimization, as we do not want the compiler to remove code blocks that we inserted. From the figure it can be seen that a task

```
..
SC_MODULE(mod) {
  void task0() { .. }
  void task1() { .. }
  ..
  SC_CTOR(mod) {
    SC_THREAD(task0);
    SC_THREAD(task1);
    ..
  }
};

int sc_main(int argc, char **argv) {
  mod m("modname");
  sc_start();
  ..
}
```

Figure 5.4. Representative SystemC synthetic benchmark example.

parallel application uses read-only, private variables and performs computations on those variables. The size of these variables are obtained by scaling down the number of those variables from the original application. The threads do not communicate among each other as there are no shared variables between the threads. The threads all are all created by the main thread. Furthermore, each thread has a unique PC because they execute separate tasks. The low-level code blocks are all added to the main function. In this case, we only added low-level code blocks to increase CMR and decrease BMR.

```
SC_MODULE(mod) {
  /* HL: general threading, PC */
  void task1() {
    int workCount;      /* work counter */
    int localMemTemp;   /* temp read/write data */
    int localMemIt;     /* temp read/write data */
    int localAddr1[1]; /* thread local mem */

    /* HL: data sharing, private */
    for (localMemIt = 0; localMemIt < 1; localMemIt++) {
      /* write local mem */
      localAddr1[localMemIt] = localMemIt;
    }

    /* HL: general threading, lifetime */
    for (workCount = 0; workCount < 1013; workCount++) {
      /* do something */
    }

    /* HL: data sharing, private */
    for (localMemIt = 0; localMemIt < 1; localMemIt++) {
      /* read local mem */
      localMemTemp = localAddr1[localMemIt];
      /* use localMemTemp */
    }
  }

  /* Code blocks for task2, task3, and task4 are similar to task1 */

  SC_CTOR(mod) {
    /* HL: general threading, PC and # threads */
    SC_THREAD(task1);
    SC_THREAD(task2);
    SC_THREAD(task3);
    SC_THREAD(task4);
  }
};
```

Figure 5.5. An example synthetic SystemC code obtained from PARSEC Blackscholes application with 4 threads showing low-level and high-level code blocks.

```
int globAddr1[197];

int sc_main(int argc, char **argv) {
  register unsigned int workCount; /* work counter */
  int globMemTemp;                 /* temp read/write data */
  int globMemIt;                   /* temp read/write data */
  mod m("modname");
  sc_start();

  /* HL: data sharing, read-only */
  for (globMemIt = 0; globMemIt < 197; globMemIt++) {
    /* read global mem */
    globMemTemp = globAddr1[globMemIt];
    /* use globMemTemp */
  }

  /* HL: general threading, lifetime */
  for (workCount = 0; workCount < 1024; workCount++) {
    /* do something */
  }

  /* LL: code block for CMR increment */
  register int wi;
  int *wa, wsize = (1024 * 1024) * 4;
  wa = (int *) malloc(wsize * sizeof(int));
  for (workCount = 0; workCount < 1159; workCount++) {
    /* do something */
    wa[rand() % wsize] = 0;
  }
  free(wa);

  /* LL: code block for BMR decrement */
  int b1 = 288, b2 = 3, bres = 0;
  for (workCount = 0; workCount < 449740; workCount++) {
    /* do something */
    if (workCount >= 0) {
      bres = b1 + b2 + (b1 / b2);
    }
  }
  return 0;
}
```

Figure 5.5. (Cont.) An example synthetic SystemC code obtained from PARSEC Blackscholes application with 4 threads showing low-level and high-level code blocks.

6. EXPERIMENTS

We present results of using our tool on various designs in this chapter. It is not possible to compare these results with other tools as none of the works in the literature generate system-level benchmarks.

First set of examples we use in our experiments are from PARSEC [5] benchmark suite that we use as Pthreads applications. PARSEC suite is composed of real world applications from different domains. Table 6.1 lists the domains [5] and parallel patterns [3] of PARSEC applications we used in our experiments. These are used to synthesize system-level SystemC applications. Second set consists of SystemC applications from various sources including the Accellera Systems Initiative SystemC/TLM distribution [25], Doulos TLM tutorial examples [33], samples from previous studies about SystemC [34, 35, 36] and our own samples written to check pattern recognition. These applications are used to synthesize Pthreads applications. We will refer to the former set of applications as Pthreads-to-SystemC (P2S) and the latter as SystemC-to-Pthreads (S2P) for convenience.

We have used three systems in our experiments to measure the portability of our synthetics. First system is where we do the synthesis hence it should have the highest accuracy of the three. In other systems use the synthetics created in the first system and compare the results with the original application. This way we can measure the portability of our synthetics, that is whether or not they keep imitating the original on different machines. Specifications of these systems are given in Table 6.2.

For the low-level characteristics we use three metrics, Instruction-per-cycle (IPC), Cache-miss-rate (CHR), and Branch-miss-rate (BMR). We expect all three metrics to have a certain amount of similarity to the original application. We used 70% for the individual similarity threshold and 80% for overall similarity threshold. We also tried to use different thresholds for SystemC applications to see how it effects the success chance of convergence. All the patterns in the synthetics match the patterns of the

Table 6.1. Domains [5] and parallel patterns [3] of PARSEC applications used in experiments.

Program	**Domain**	**Parallel Pattern**
facesim	Animation	Task Parallel
fluidanimate	Animation	Geometric Decomposition
bodytrack	Computer Vision	Geometric Decomposition
blackscholes	Financial Analysis	Task Parallel
swaptions	Financial Analysis	Task Parallel
ferret	Similarity Search	Pipeline

Table 6.2. Hardware and software specifications of the platforms we use in our experiments.

	System-I	**System-II**	**System-III**
# Cores	2x4	2	2x8
# Procs	2x8	4	2x16
DRAM	32GB	4GB	32GB
L1	2x256kb, 8-way	64kb, 8-way	2x512kb, 8-way
L2	2x1024kb, 8-way	256kb, 8-way	2x2048kb, 8-way
LLC (L3)	2x8192kb, 16-way	3072kb, 12-way	2x20480kb
Architecture	x86-64	x86-64	x86-64
Processor	Intel Xeon E5520	Intel Core i5-246	Intel Xeon E5-2680
Kernel	Linux 3.11.0-17	Linux 3.11.0-19	Linux 3.7.10-1.16
Distribution	Ubuntu 12.04	Ubuntu 12.04	OpenSUSE 12.3

original applications.

After the synthesis we use hit rates to calculate similarities for caching and braching. Otherwise they would yield too much error since values are usually very small especially for branch-miss-rate (BMR). Overall similarity is calculated by taking the average of errors in Instruction-per-cycle (IPC), Cache-hit-rate (CHR) and Branch-hit-rate (BHR).

Since some of the examples take too long to converge and possibly do not converge at all we have used thresholds to interrupt the synthesis in the middle. We have used a timeout of 1 hour or 200 iterations as the threshold whichever comes first. Linux perf tools is set to take the average of 10 runs.

For SystemC implementation we used version 2.3.0 provided by Accellera Systems Initiative [25]. This implementation maps SystemC threads to either Pthreads or QThreads. We have used the library with Pthreads option enabled in our study.

6.1. Pthreads-to-SystemC (P2S)

In the first set of experiments, we used PARSEC applications to create the synthetics. We have selected 6 applications from the suite that we were able to successfully create the synthetics. In other applications we either had problems in instrumentation or convergence with the evolutionary algorithm. We used simmedium inputs for the results shown here.

6.1.1. IPC Similarities in P2S Experiments

Figure 6.1a shows IPC values for the first hardware in which the synthesis is made. Average IPC error is measured to be 15% with a minimum of 3% for `ferret` and a maximum of 23% for `blackscholes`.

Figure 6.1b shows IPC values measured in the second hardware. Average IPC

error is measured to be 25% with a minimum of 13% for `blackscholes` and a maximum of 54% for `ferret`. 5 of 6 applications other than `ferret` seem to have an error percentage below 30% with the exception of `swaptions` having 30.5%.

Figure 6.1c shows IPC values measured in the third hardware. Average IPC error is measured to be 33% with a minimum of 8% for `ferret` and a maximum of 51% for `bodytrack` and `blackscholes`. Only `facesim`, `fluidanimate` and `ferret` seem to have an error percentage below 30% with the exception of the first one having 30.1%.

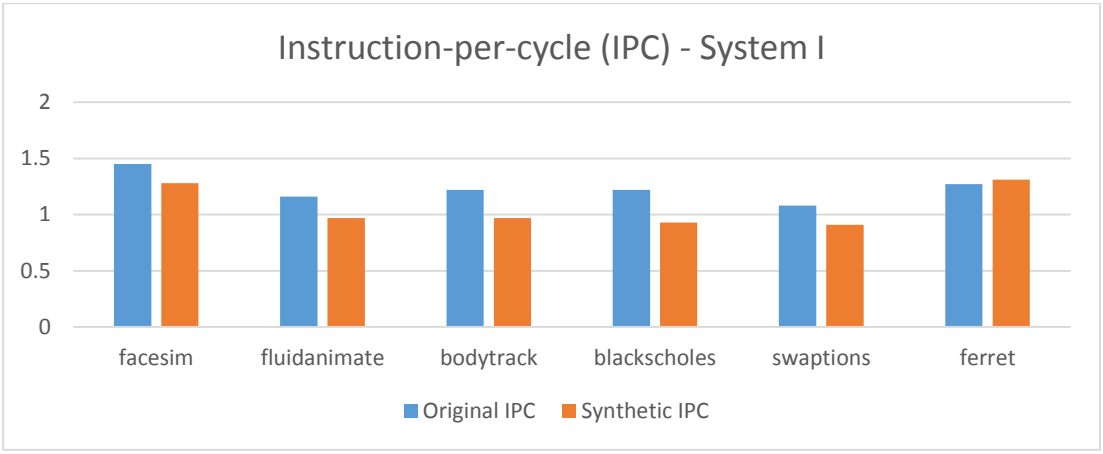
Original and synthetic applications running on these systems have an average standard deviation of 0.21 with a minimum of 0.16 for `facesim` and a maximum of 0.29 for `bodytrack`. IPC errors starts with an average on 15% in the original hardware and goes up to 25% in the second hardware and slightly pass the threshold of 30% in the third one. These values suggest that there is not much difficulty in matching the IPC value of the synthetic with the original in the first hardware although there is still a room for improvement required for the portability.

6.1.2. CHR Similarities in P2S Experiments

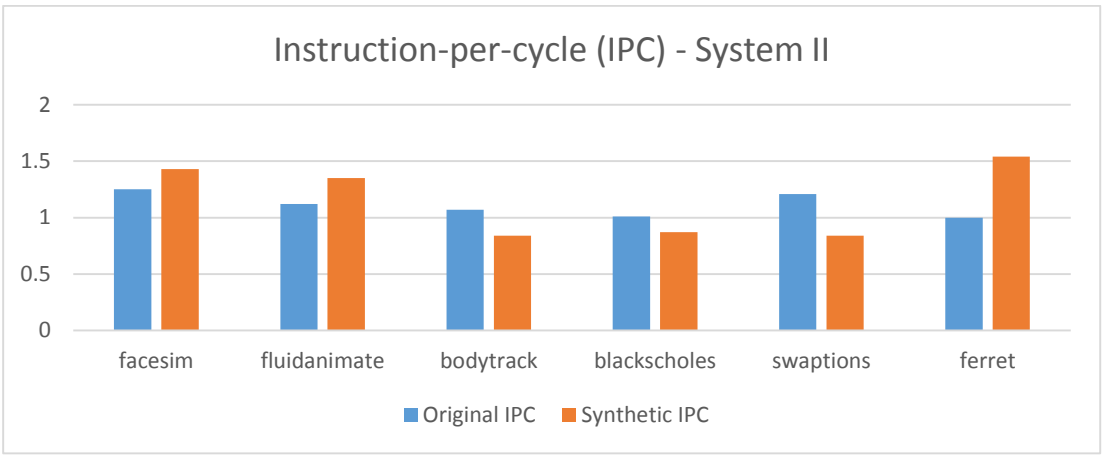
Figure 6.2a shows CHR values for the first hardware in which the synthesis is made. Average CHR error is measured to be 4% with a minimum of 0.4% for `swaptions` and a maximum of 12% for `fluidanimate`.

Figure 6.2b shows CHR values measured in the second hardware. Average CHR error is measured to be 8% with a minimum of 0.2% for `facesim` and a maximum of 17% for `bodytrack`. All of the applications seem to have an error percentage below 30%.

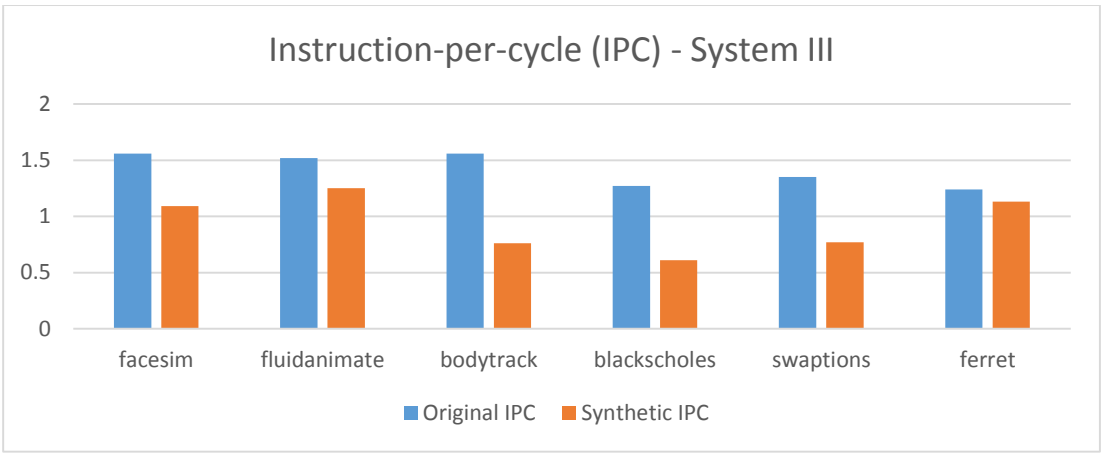
Figure 6.2c shows CHR values measured in the third hardware. Average CHR error is measured to be 23% with a minimum of 6% for `swaptions` and a maximum of 38% for `facesim`. 4 of 6 applications other than `facesim` and `blackscholes` seem to have an error percentage below 30%.



(a)

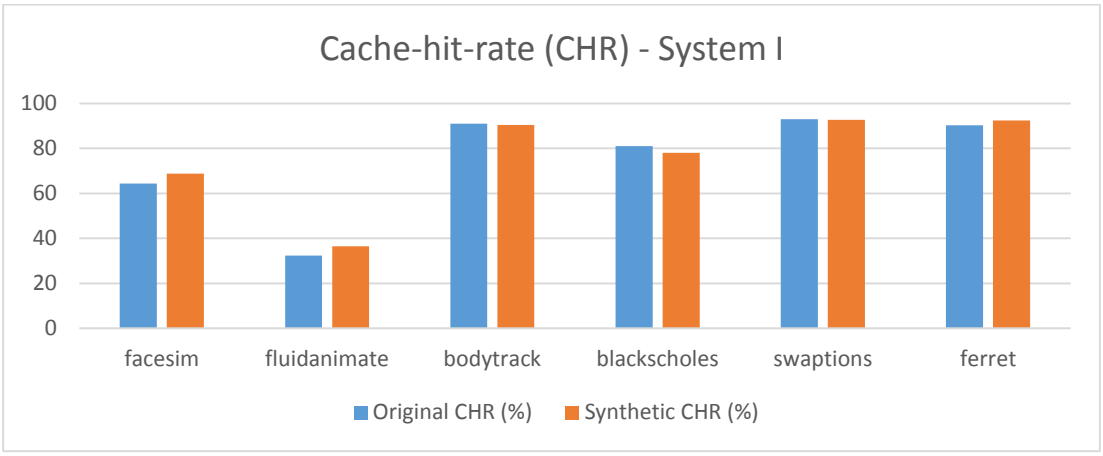


(b)

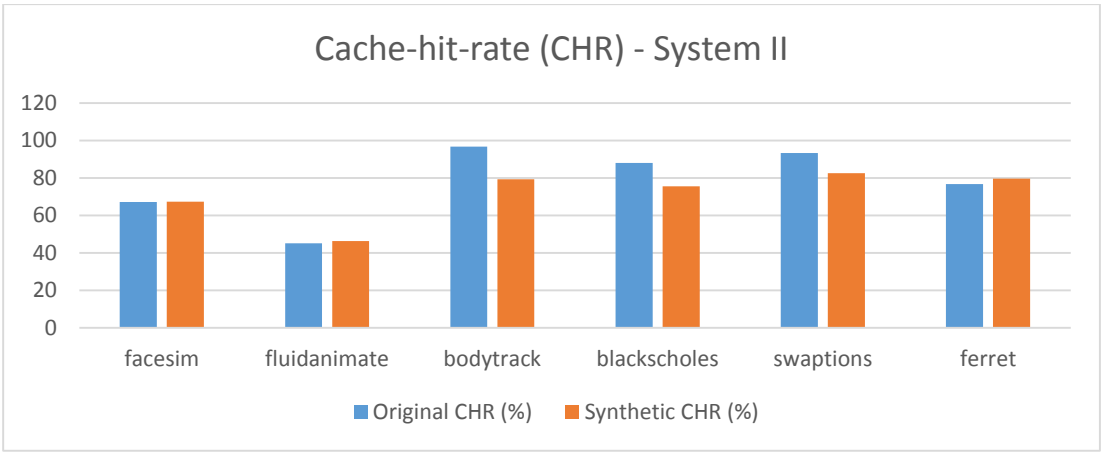


(c)

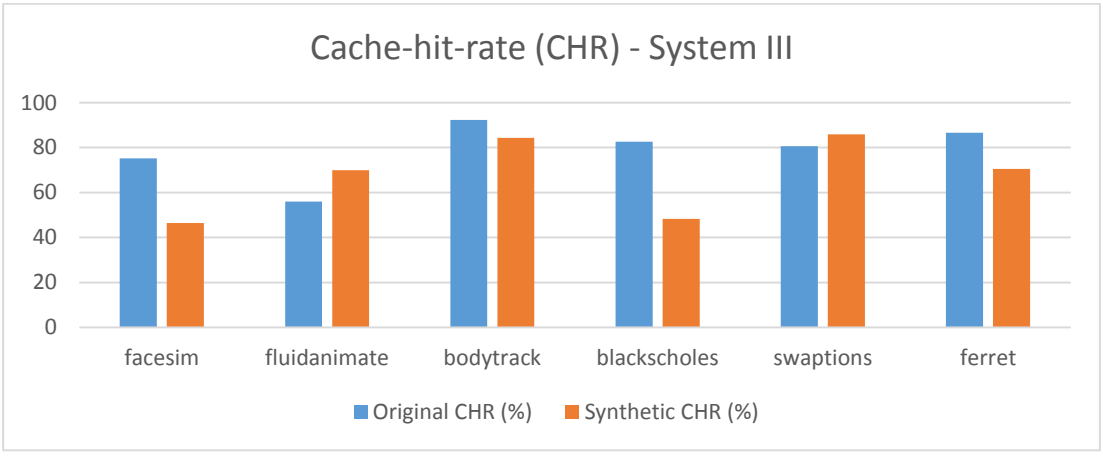
Figure 6.1. Instruction-per-cycle values of original (PARSEC) and synthetic applications created with 70% individual and 80% overall similarities thresholds on (a) System-I, (b) System-II and (c) System-III.



(a)



(b)



(c)

Figure 6.2. Cache-hit-rate values of original (PARSEC) and synthetic applications created with 70% individual and 80% overall similarities thresholds on (a) System-I, (b) System-II and (c) System-III.

Original and synthetic applications running on these systems have an average standard deviation of 9% with a minimum of 5% for `swaptions` and a maximum of 14 for `blackscholes`. CHR errors starts with an average on 4% in the original hardware and goes up to 8% in the second hardware and 23% in the third one. First and second systems seem to be working while there are some examples in the third one that misses the threshold of 30%. These are namely `facesim` and `blackscholes`, both having a dramatic decrease in CHR in the third system. This is likely due to different cache algorithms used in these systems since cache sizes are already the highest of the three for this machine. Nevertheless these misses are still close to the threshold and CHR error values in general seem to be in acceptable ranges.

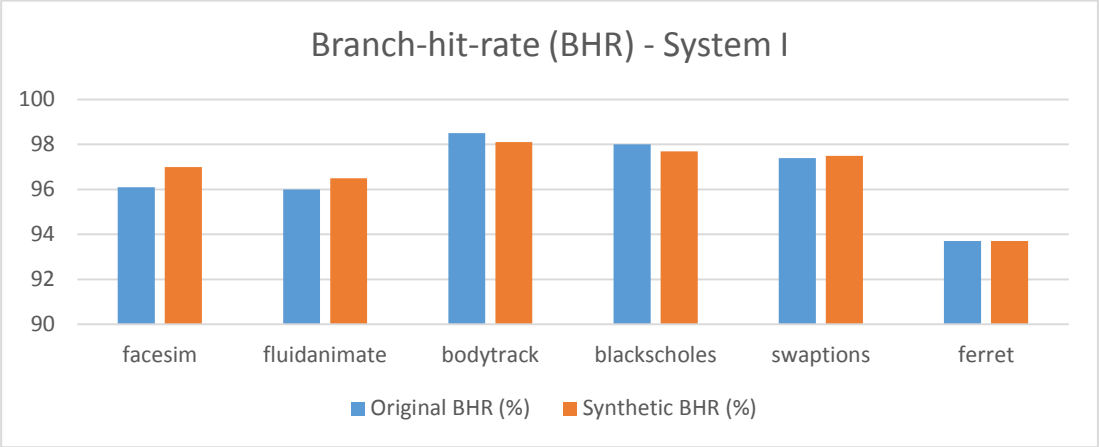
6.1.3. BHR Similarities in P2S Experiments

Figure 6.3a shows BHR values for the first hardware in which the synthesis is made. Average BHR error is measured to be 0.3% with a minimum of 0% for `ferret` and a maximum of 0.9% for `facesim`.

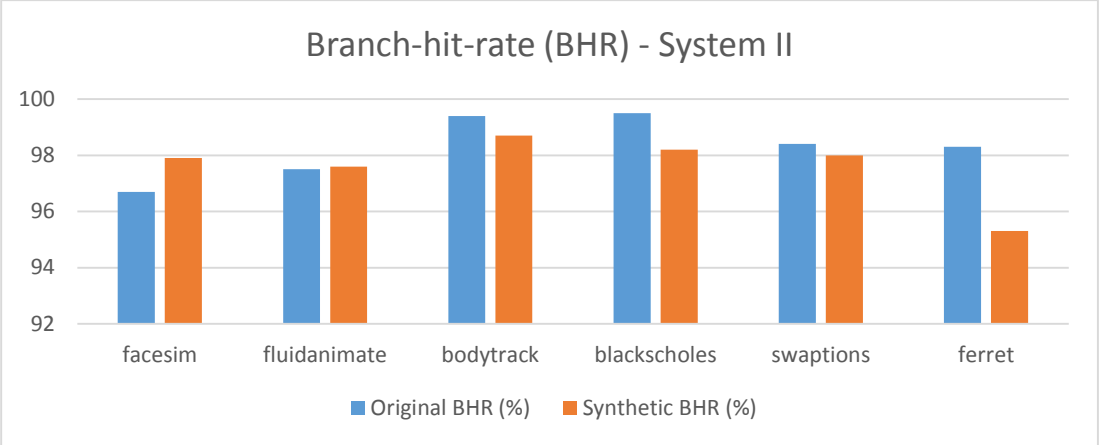
Figure 6.3b shows BHR values measured in the second hardware. Average BHR error is measured to be 1.1% with a minimum of 0.1% for `fluidanimate` and a maximum of 3% for `ferret`. All of the applications have an error percentage below 30%.

Figure 6.3c shows BHR values measured in the third hardware. Average BHR error is measured to be 0.5% with a minimum of 0.1% for `facesim` and `ferret` and a maximum of 1.6% for `blackscholes`. All of the applications have an error percentage below 30%.

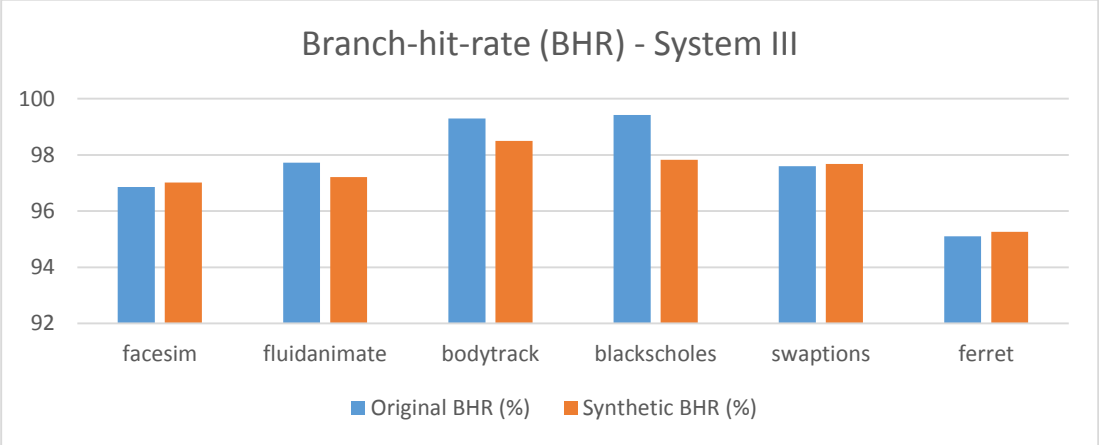
Original and synthetic applications running on these systems have an average standard deviation of 0.7% with a minimum of 0.3% for `swaptions` and a maximum of 1.6% for `ferret`. BHR error values are relatively small compared to other characteristics with a maximum of 3% in all three systems. This is due to the fact that we use miss rate for the synthesis and hit rate for the error measurement. Since branch predictions almost always have an accuracy above 90%, miss rates yield much higher



(a)



(b)



(c)

Figure 6.3. Branch-hit-rate values of original (PARSEC) and synthetic applications created with 70% individual and 80% overall similarities thresholds on (a) System-I, (b) System-II and (c) System-III.

values than hit rates in error calculations.

6.1.4. Overall Similarities in P2S Experiments

Figure 6.4a shows overall similarities running on first hardware in which the synthesis is made. Overall error percentages of low-level characteristics have an average of 6% ranging from 1% to 9%.

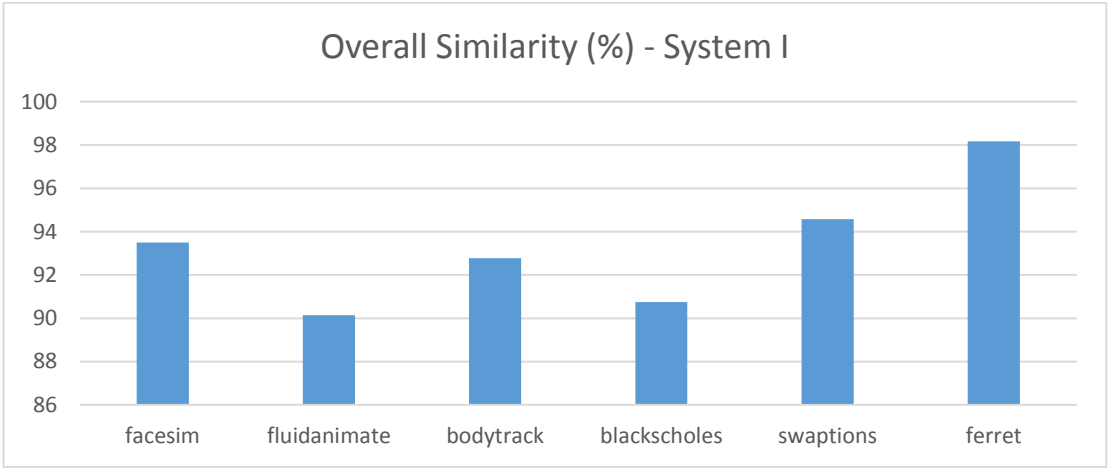
Figure 6.4b shows overall similarities running on the second hardware. Overall error percentages have an average of 11% ranging from 5% to 20%. All samples have an error percentage below 20% with the exception of `ferret` having 20.2%.

Figure 6.4c shows overall similarities running on the third hardware. Overall error percentages have an average of 19% ranging from 9% to 31%. 3 of 6 applications have an error percentage below 20% with two close results, `facesim` having 22.8% and `bodytrack` having 20.2%, and an acceptable result, `blackscholes` having 31.6%.

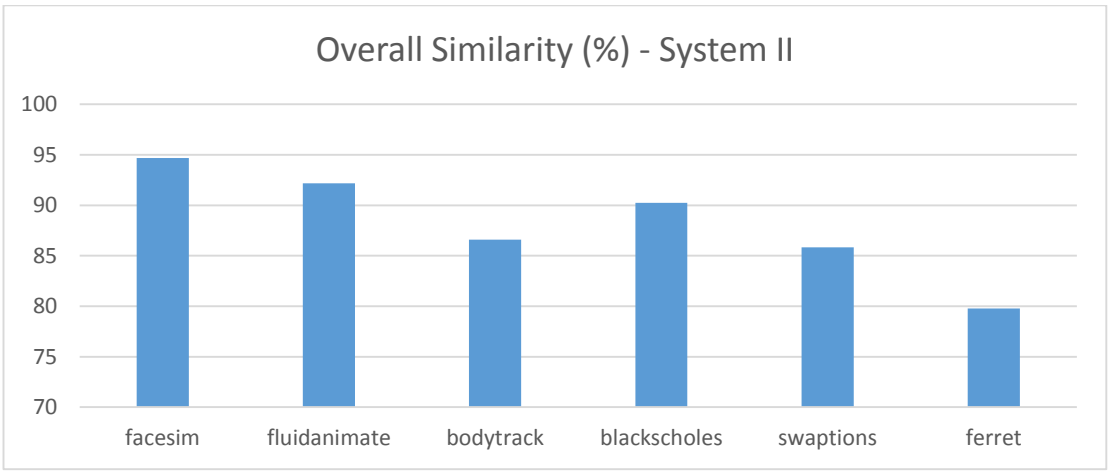
Overall errors starts with an average of 6% in the original hardware and goes up to 11% in the second hardware and 19% in the third one. Despite the averages all being under the threshold of 20%, there are still some examples exceeding this value in the third system. Out of 3 such applications, one of them is pretty close to the threshold while the other two applications are the ones which have problems in matching CHR. We can infer that the synthesis is successful while the portability is acceptable but still can take advantage of some possible improvements.

6.2. SystemC-to-Pthreads (S2P)

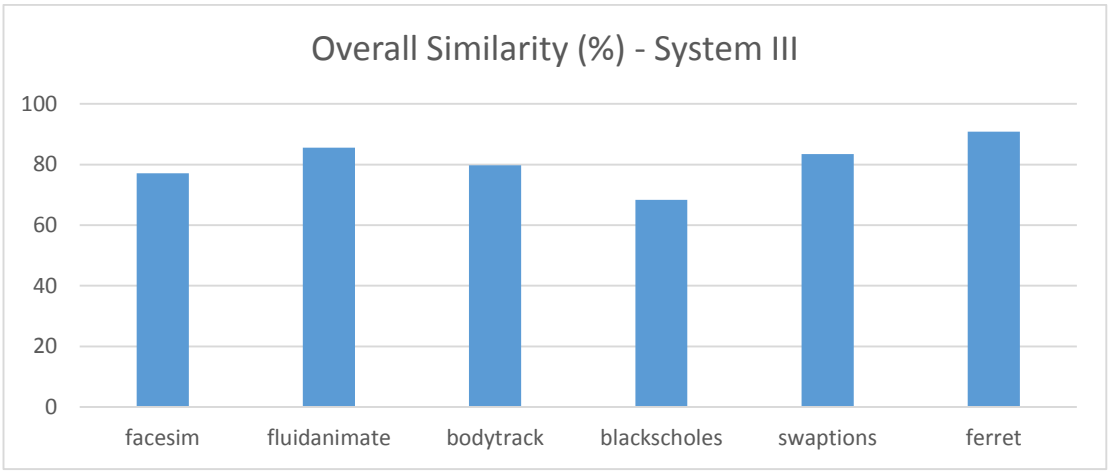
In the second set of experiments, we used SystemC and TLM applications to create the synthetics. We show here 35 applications from our collection that we were able to successfully create the synthetics.



(a)



(b)



(c)

Figure 6.4. Overall similarities of original (PARSEC) and synthetic applications created with 70% individual and 80% overall similarities thresholds on (a) System-I, (b) System-II and (c) System-III.

6.2.1. IPC Similarities in S2P Experiments

Figures 6.5a and 6.5d show IPC values for the first hardware in which the synthesis is made. Average IPC error is measured to be 11% with a minimum of 0% and a maximum of 24%.

Figures 6.5b and 6.5e show IPC values measured in the second hardware. Average IPC error is measured to be 22% with a minimum of 6% and a maximum of 49%. 31 of 35 applications seem to have an error percentage below 30%.

Figures 6.5c and 6.5f show IPC values measured in the third hardware. Average IPC error is measured to be 24% with a minimum of 3% and a maximum of 63%. 29 of 35 applications seem to have an error percentage below 30%.

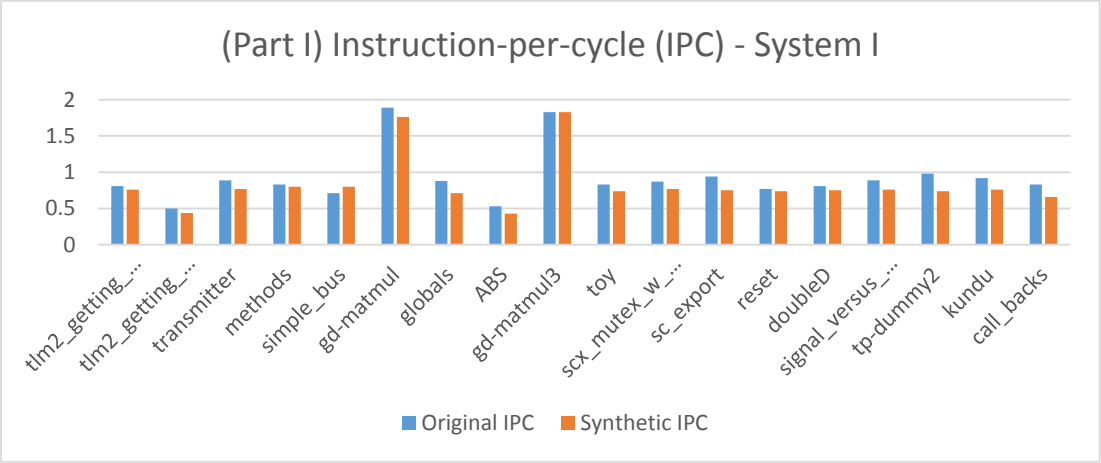
Original and synthetic applications running on these systems have an average standard deviation of 0.13 with a minimum of 0.05 and a maximum of 0.32. IPC errors starts with an average on 11% in the original hardware and goes up to 22% in the second hardware and 24% in the third one. Error percentages mostly fall below the threshold of 30% suggesting that the portability is assured.

6.2.2. CHR Similarities in S2P Experiments

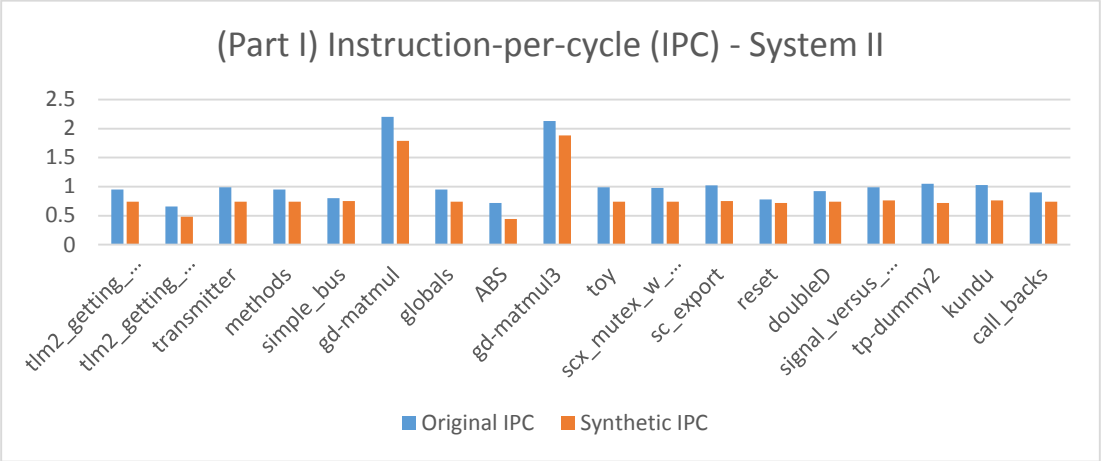
Figures 6.6a and 6.6d show CHR values for the first hardware in which the synthesis is made. Average CHR error is measured to be 1.8% with a minimum of 0.2% and a maximum of 7%.

Figures 6.6b and 6.6e show CHR values measured in the second hardware. Average CHR error is measured to be 14% with a minimum of 2% and a maximum of 24%. All of the applications seem to have an error percentage below 30%.

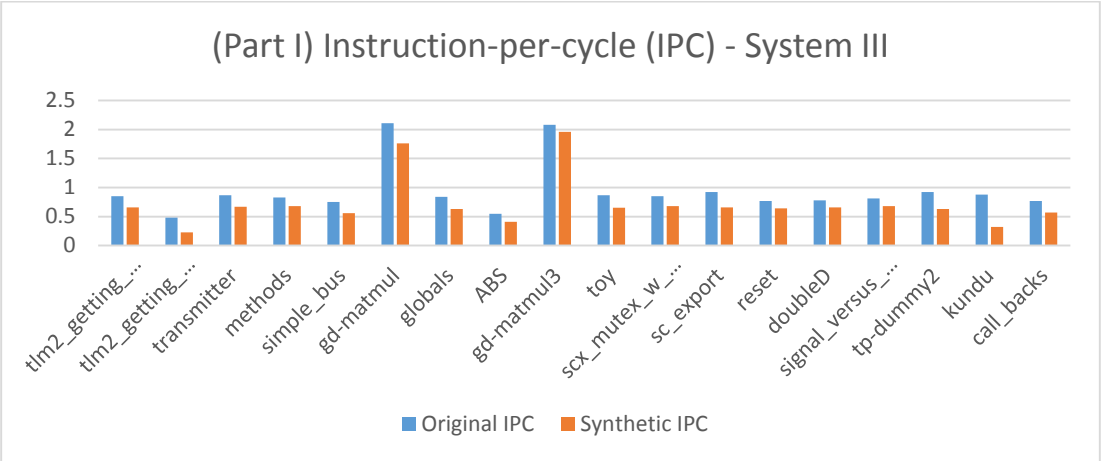
Figures 6.6c and 6.6f show CHR values measured in the third hardware. Average CHR error is measured to be 7% with a minimum of 0.2% and a maximum of 48%. 33



(a)

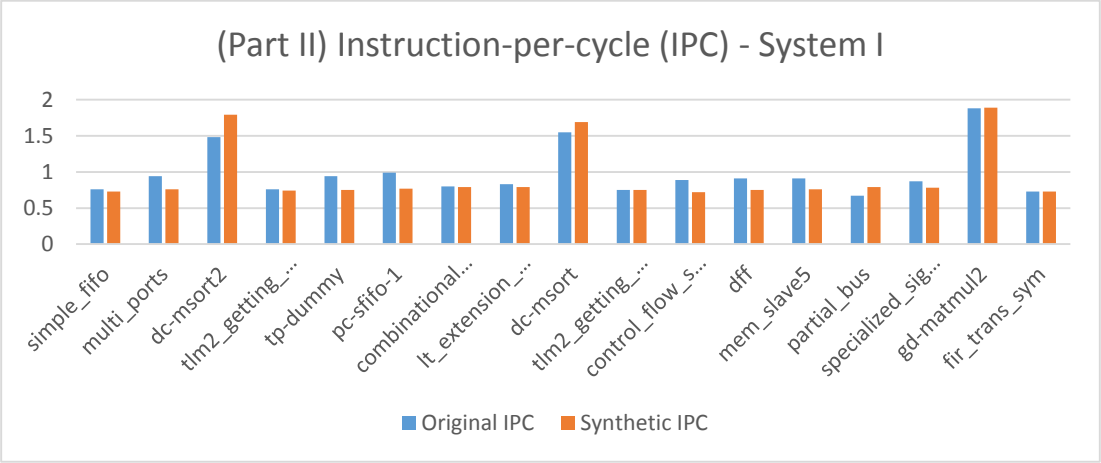


(b)

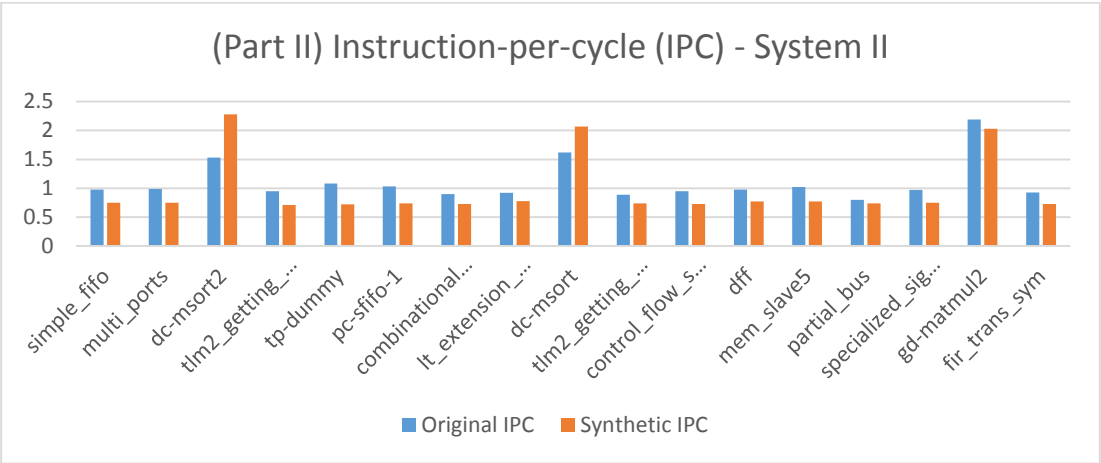


(c)

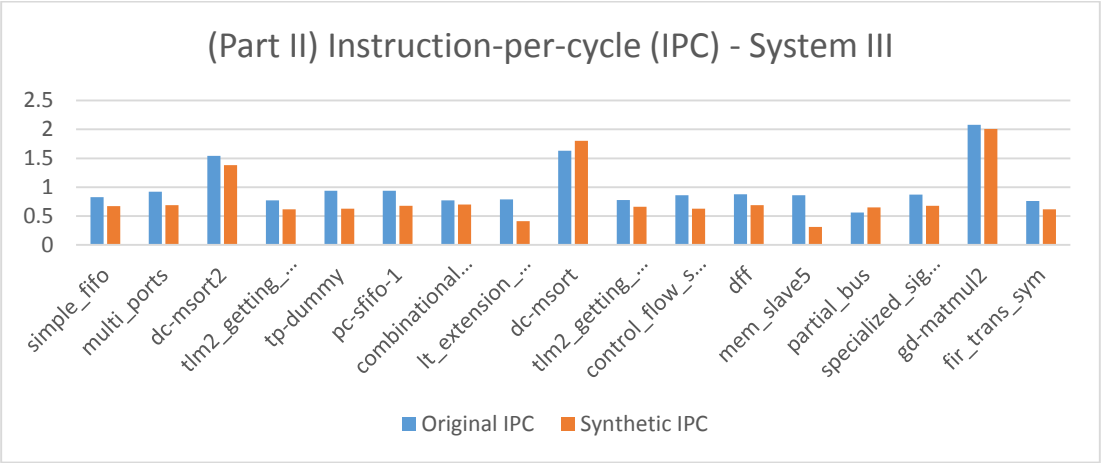
Figure 6.5. Instruction-per-cycle values of original (SystemC) and synthetic applications created with 70% individual and 80% overall similarities thresholds on (a) System-I, (b) System-II and (c) System-III.



(d)



(e)



(f)

Figure 6.5. (Cont.) Instruction-per-cycle values of original (SystemC) and synthetic applications created with 70% individual and 80% overall similarities thresholds on (d) System-I, (e) System-II and (f) System-III.

of 35 applications seem to have an error percentage below 30%.

Original and synthetic applications running on these systems have an average standard deviation of 6% with a minimum of 3% and a maximum of 17%. CHR errors starts with an average of 1.8% in the original hardware and goes up to 14% in the second hardware and 7% in the third one. Even though the average error percentage is lower in the third system compared to the second one, there are still some applications which have an error percentage above the threshold of 30% in the third system while there is none in the second one. This is again caused by a dramatic decrease in CHR in the third machine possibly due to the use of different cache algorithms. These are considered outliers since there are only 2 examples that exceed the threshold.

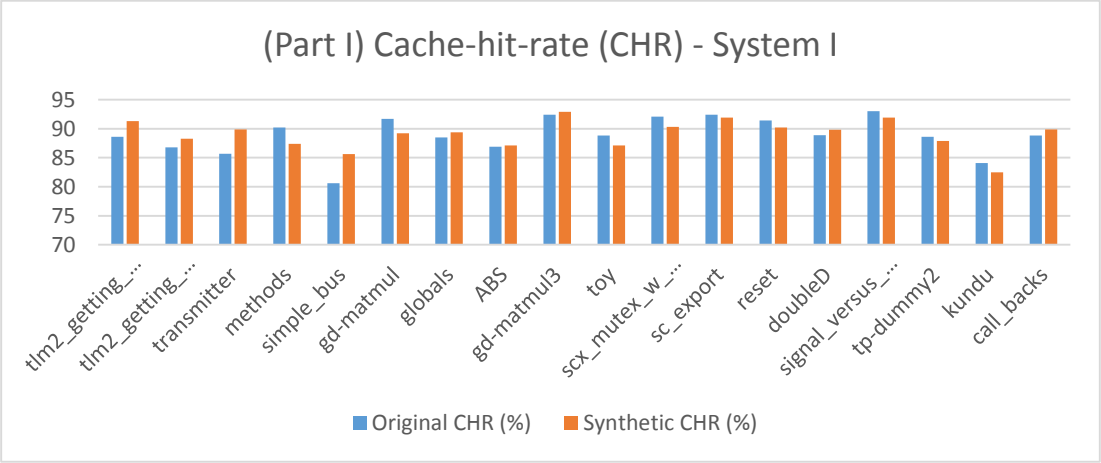
6.2.3. BHR Similarities in S2P Experiments

Figures 6.7a and 6.7d show BHR values for the first hardware in which the synthesis is made. Average BHR error is measured to be 0.5% with a minimum of 0.1% and a maximum of 1.2%.

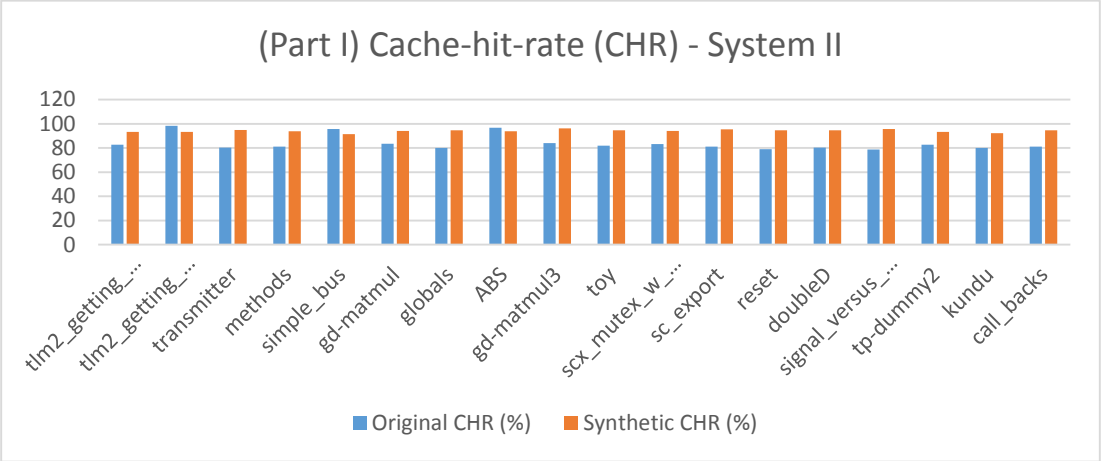
Figures 6.7b and 6.7e show BHR values measured in the second hardware. Average BHR error is measured to be 0.7% with a minimum of 0.1% and a maximum of 1.8%. All of the applications have an error percentage below 30%.

Figures 6.7c and 6.7f show BHR values measured in the third hardware. Average BHR error is measured to be 0.7% with a minimum of 0% and a maximum of 1.9%. All of the applications have an error percentage below 30%.

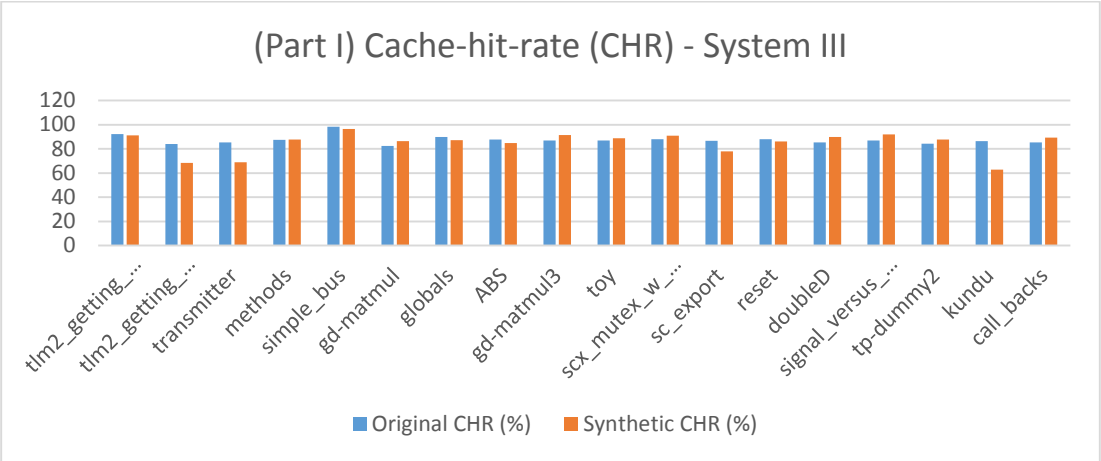
Original and synthetic applications running on these systems have an average standard deviation of 0.5% with a minimum of 0.2% and a maximum of 1%. BHR error values are again quite small being at most 2% in all three systems due to the use of hit rate in error calculations instead of miss rate as explained before.



(a)

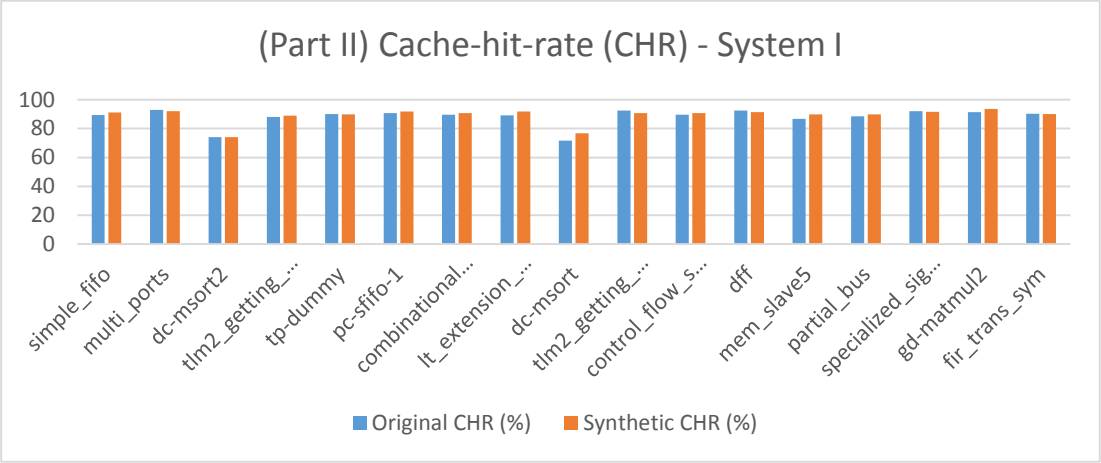


(b)

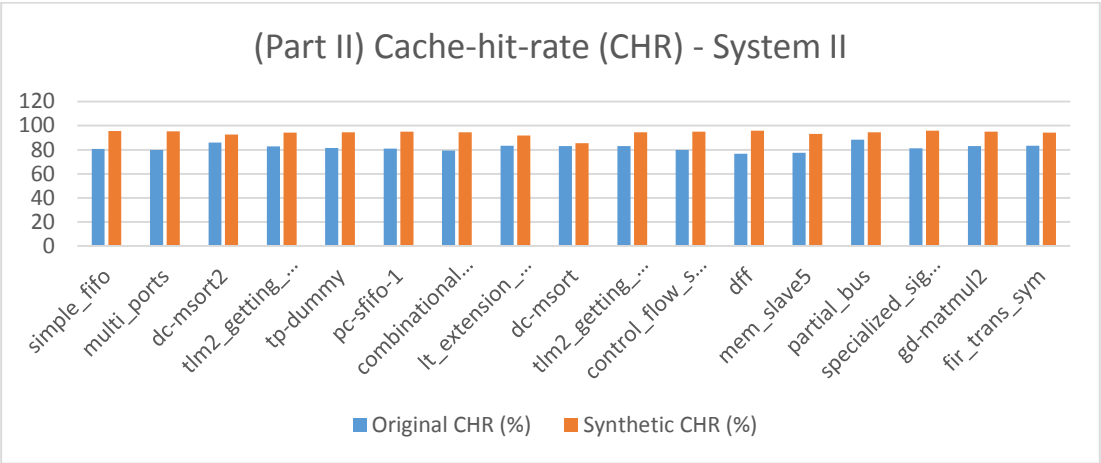


(c)

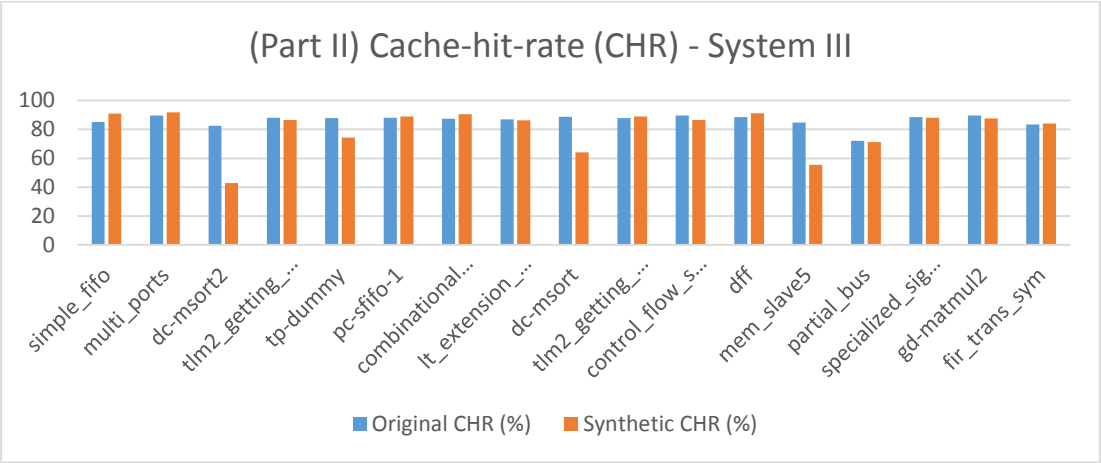
Figure 6.6. Cache-hit-rate values of original (SystemC) and synthetic applications created with 70% individual and 80% overall similarities thresholds on (a) System-I, (b) System-II and (c) System-III.



(d)

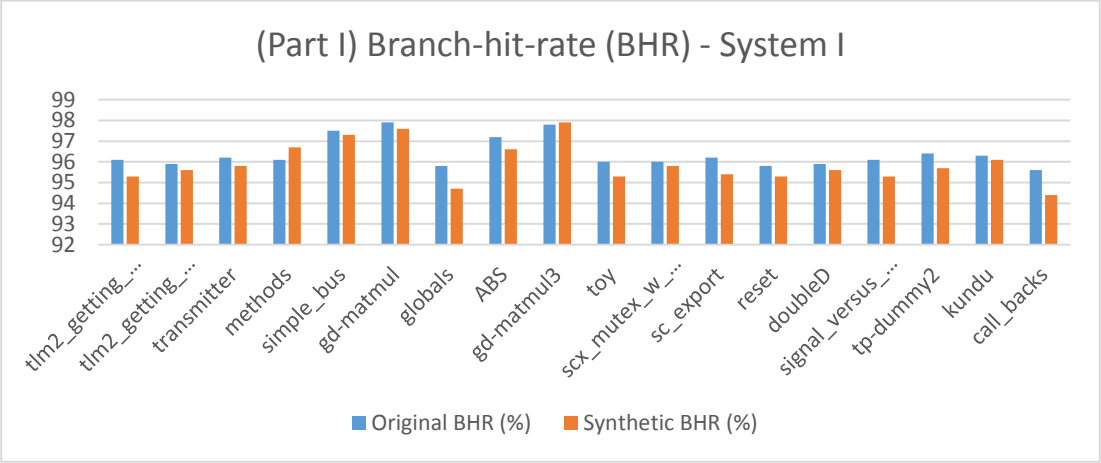


(e)

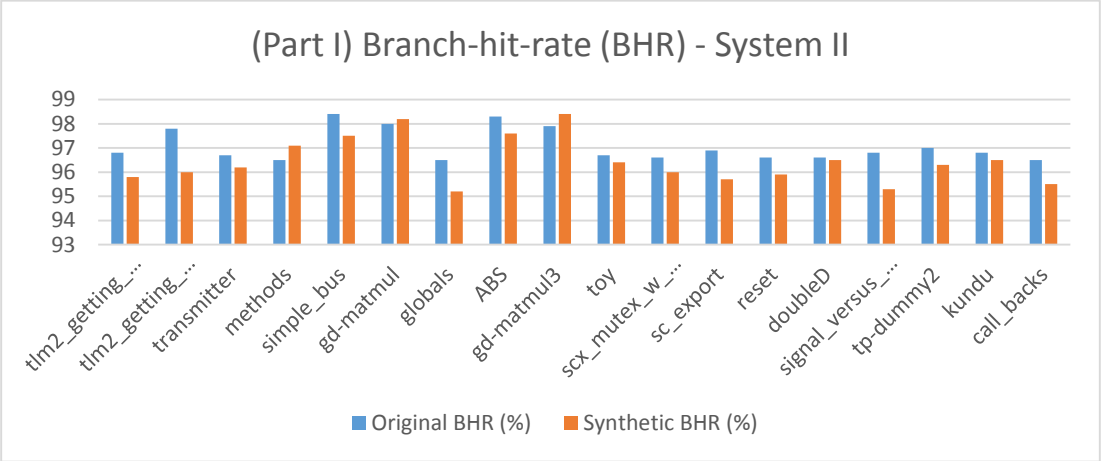


(f)

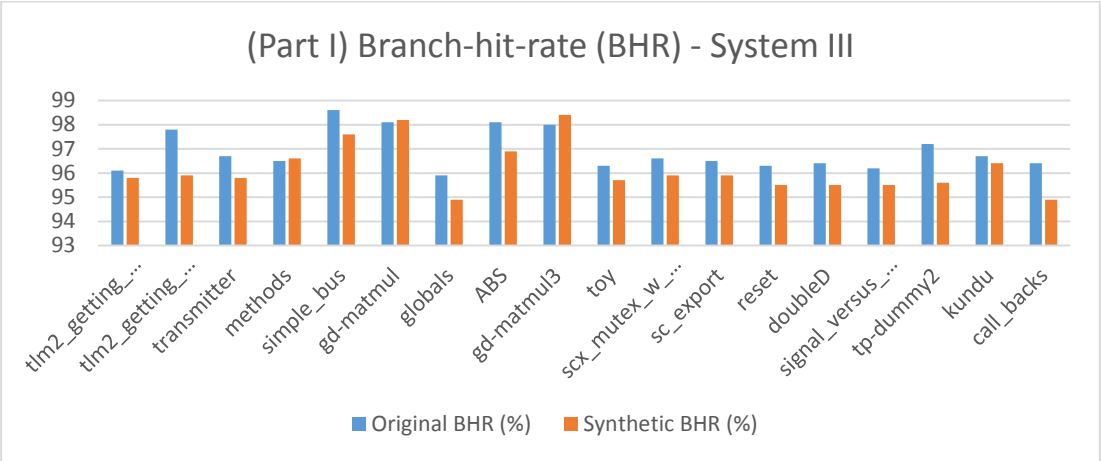
Figure 6.6. (Cont.) Cache-hit-rate values of original (SystemC) and synthetic applications created with 70% individual and 80% overall similarities thresholds on (d) System-I, (e) System-II and (f) System-III.



(a)

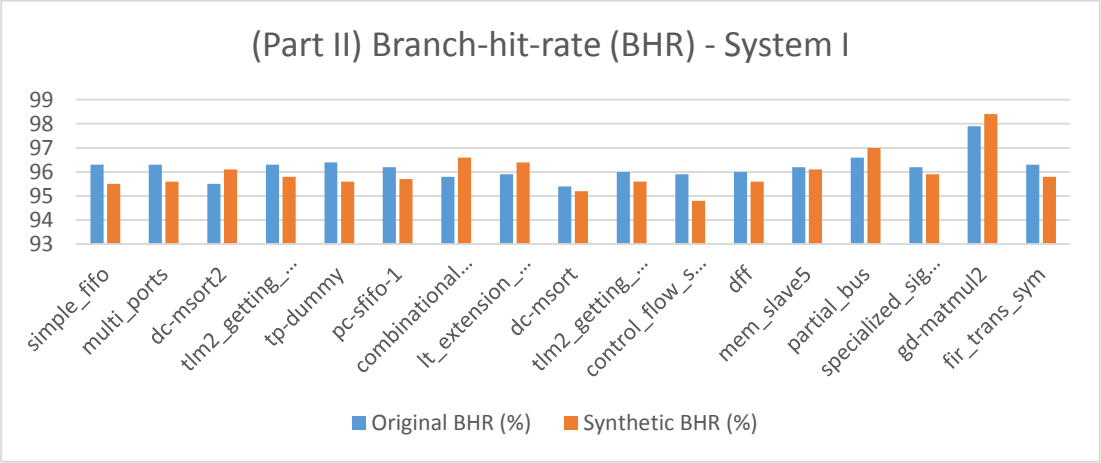


(b)

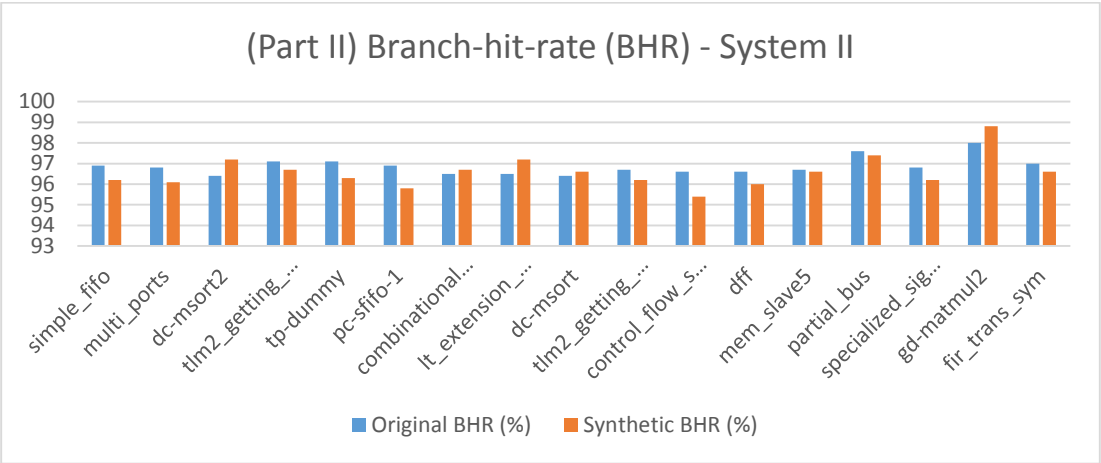


(c)

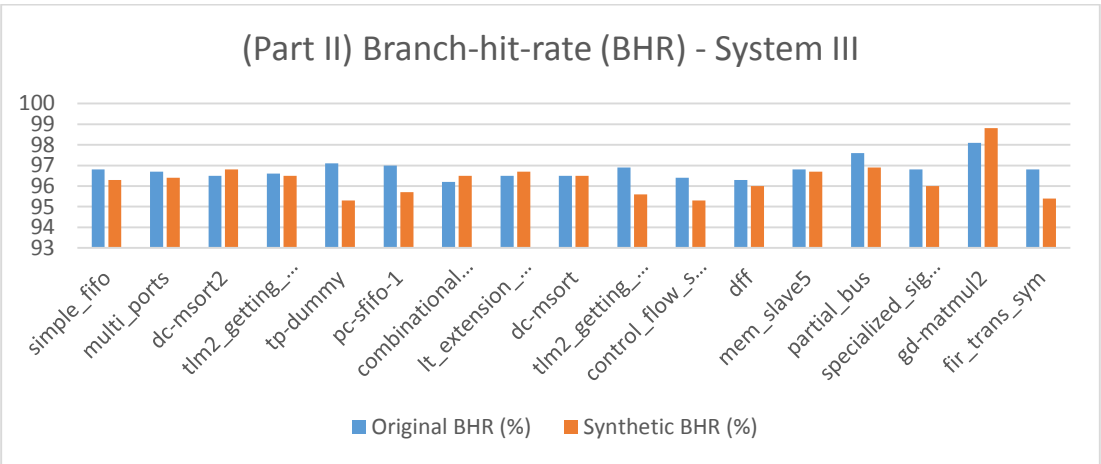
Figure 6.7. Branch-hit-rate values of original (SystemC) and synthetic applications created with 70% individual and 80% overall similarities thresholds on (a) System-I, (b) System-II and (c) System-III.



(d)



(e)



(f)

Figure 6.7. (Cont.) Branch-hit-rate values of original (SystemC) and synthetic applications created with 70% individual and 80% overall similarities thresholds on (d) System-I, (e) System-II and (f) System-III.

6.2.4. Overall Similarities in S2P Experiments

Figures 6.8a and 6.8d show overall similarities running on first hardware in which the synthesis is made. Overall error percentages of low-level characteristics have an average of 4% ranging from 0.2% to 8%.

Figures 6.8b and 6.8e show overall similarities running on the second hardware. Overall error percentages have an average of 12% ranging from 3% to 19%. All samples have an error percentage below 20%.

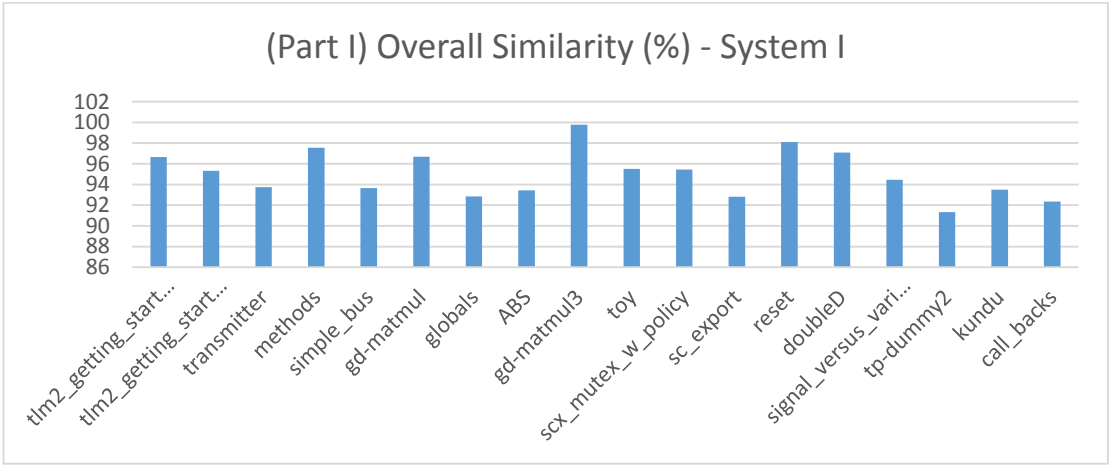
Figures 6.8c and 6.8f show overall similarities running on the third hardware. Overall error percentages have an average of 10% ranging from 2% to 32%. 32 of 35 applications have an error percentage below 20%.

Overall errors starts with an average of 4% in the original hardware and goes up to 12% in the second hardware and 10% in the third one. Only 3 applications miss the threshold of 20% in the third hardware. These applications are the ones which can not make up for their high error percentages in CHR with their error percentages in IPC and BHR. These numbers suggest that highly accurate and portable synthetics are possible with a very high probability.

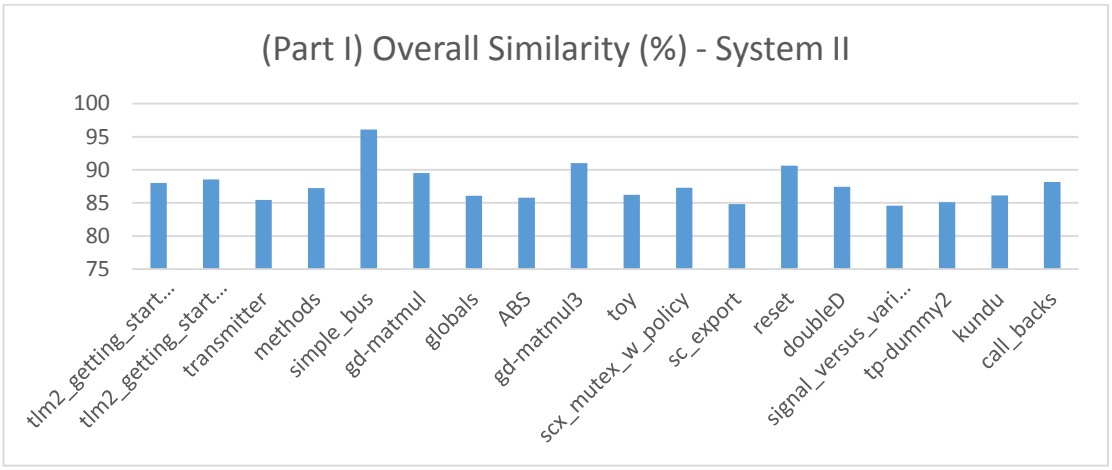
6.3. Impact of High-Level Metrics on Low-Level Metrics

We performed some experiments to see how much high-level metrics effect the similarity of low-level metrics in our synthetics. For this purpose, we deliberately tried to synthesize our benchmarks with wrong parallel software patterns. Since we only have the actual parallel patterns of PARSEC applications we used these in these experiments.

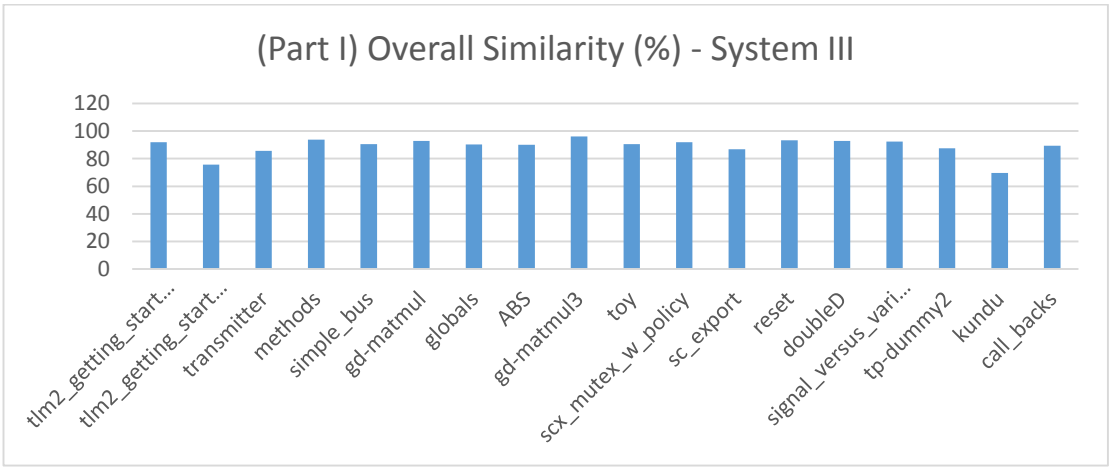
We used the same input sets used in the rest of the experiments as well as the same similarity and timeout thresholds. We observed that the convergence success percentage of our synthesis dropped dramatically in these experiments. Only for 2 of 8



(a)

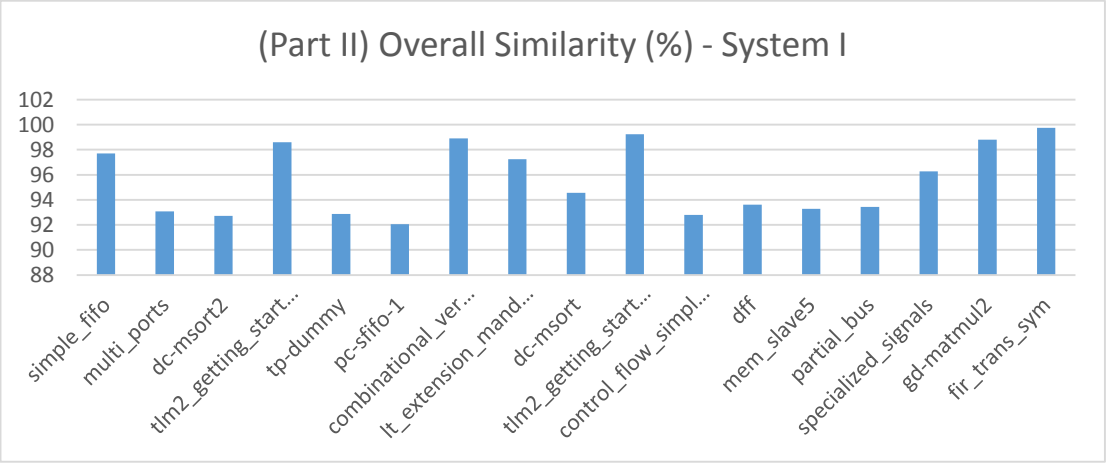


(b)

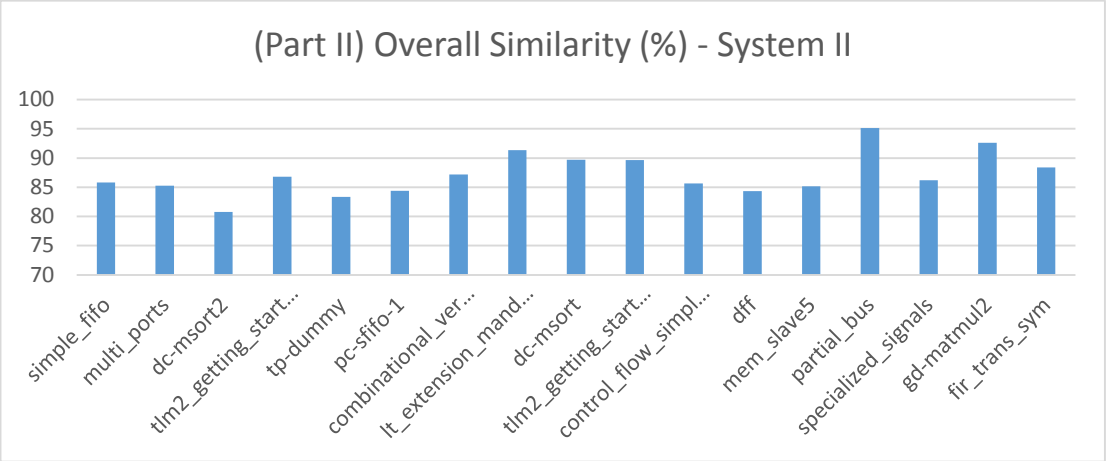


(c)

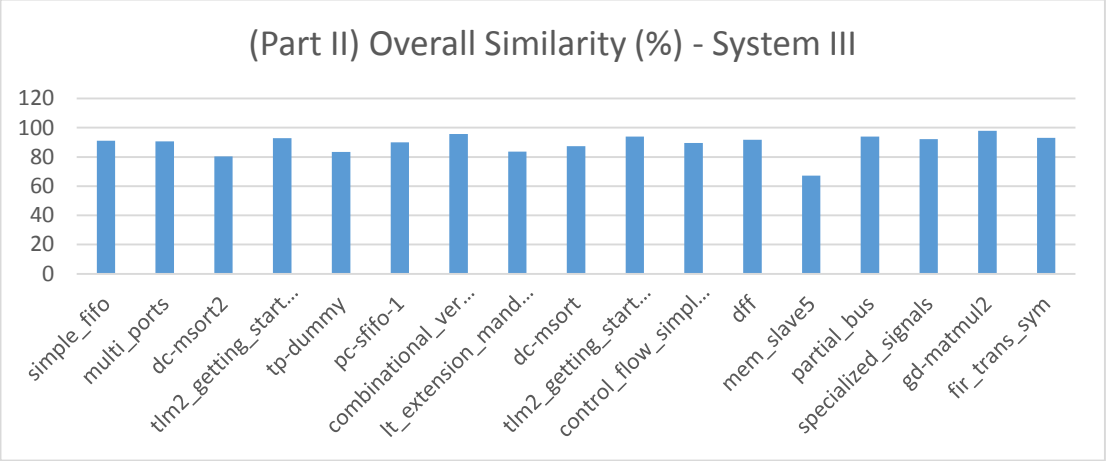
Figure 6.8. Overall similarities of original (SystemC) and synthetic applications created with 70% individual and 80% overall similarities thresholds on (a) System-I, (b) System-II and (c) System-III.



(d)



(e)



(f)

Figure 6.8. (Cont.) Overall similarities of original (SystemC) and synthetic applications created with 70% individual and 80% overall similarities thresholds on (d) System-I, (e) System-II and (f) System-III.

applications, namely `blackscholes` and `swaptions`, we were able to generate synthetic versions with close enough similarities. Therefore we confirm in these experiments that our high-level parallel software patterns are important for the convergence success percentage of our evolutionary algorithm during synthesis.

6.4. Impact of Input Changes on Synthetics

Picking a representative input set demonstrating the performance characteristics of the application is an important decision [37]. Different inputs have a potential to change the behavior of the application. Depending on the inputs used in originals our framework can generate different synthetics.

For PARSEC applications there are different input sets for each application. Test and simdev inputs are for testing purposes. Native inputs are real-world inputs that lasts about 15 minutes of execution on a regular machine. Since there is also an overhead of binary instrumentation it was not feasible to use these inputs in our experiments. Other inputs are simsmall, simmedium and simlarge that corresponds to execution times of 1 second, 5 seconds and 15 seconds on average. We opted for simmedium inputs in our experiments.

In order to test the effects of input sizes we also carried on experiments with simsmall inputs and compare these with our results. First we compared the low-level characteristics of the original applications running on simsmall and simmedium inputs. In the second part we compared low-level characteristics of the originals running on simsmall inputs with the ones synthesized from originals running on simmedium inputs. This corresponds to the case where we try to use synthetics synthesized from a particular input set as proxies for others. In the worst case, we expect to have an error which amounts to the total of the difference in the first part and the tolerance we use in the synthesis. All the experiments in this section are carried out on the first hardware.

6.4.1. Originals Running on Simsmall Input Set Compared to Originals Running on Simmedium Input Set

In this section we compare IPC, CMR and BMR values of originals running on simsmall input set with the same originals but this time running on simmedium input set. The purpose of this experiment is to understand how sensitive PARSEC applications are to the changes in inputs used in terms of performance characteristics.

Figure 6.9a shows IPC values of executions. Change in values have an average of 3% ranging from 0% for `facesim` and `blackscholes` to 12% for `swaptions`.

Figure 6.9b shows CHR values of executions. Change in values have an average of 5% ranging from 0.1% for `swaptions` to 7% for `blackscholes`.

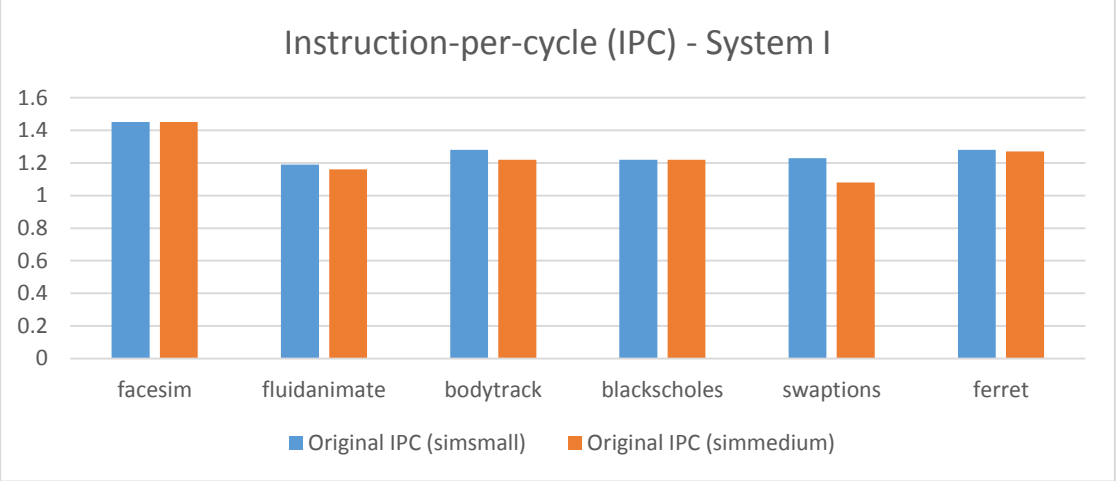
Figure 6.9c shows BHR values of executions. Change in values have an average of 0.1% ranging from 0% to 0.6%.

Figure 6.10 shows overall similarity of executions. Similarities have an average of 97% ranging from 92% for `fluidanimate` to 99% for `facesim` and `ferret`.

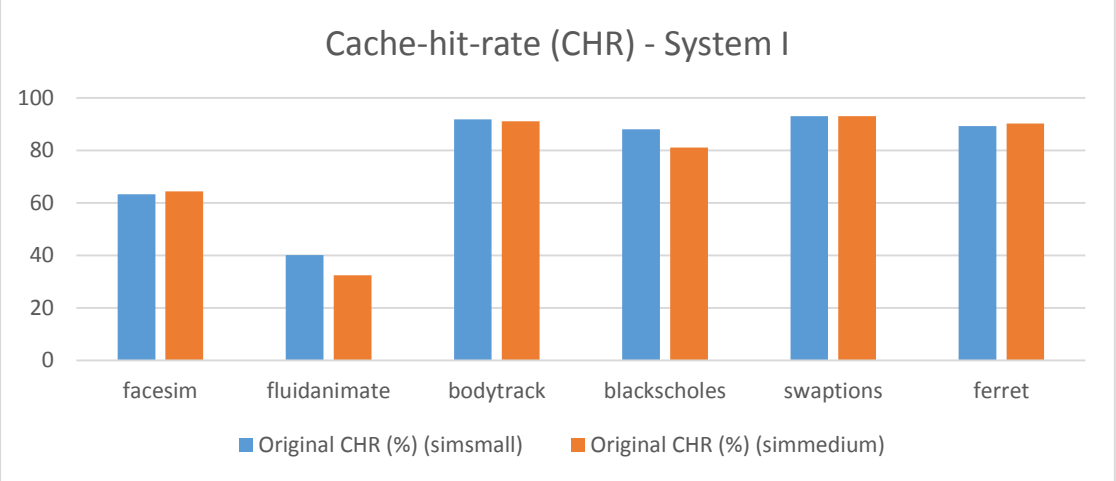
According to these results IPC is the most variational metric followed by CHR and then BHR. However these values also show that performance characteristics do not change much from one input set to another in the PARSEC benchmark suite. Therefore synthetics generated from one input set may also be used as a proxy for another input set. In the second part of our experiment we test this hypothesis.

6.4.2. Originals Running on Simsmall Input Sets Compared to Synthetics Generated from Originals Running on Simmedium Input Sets

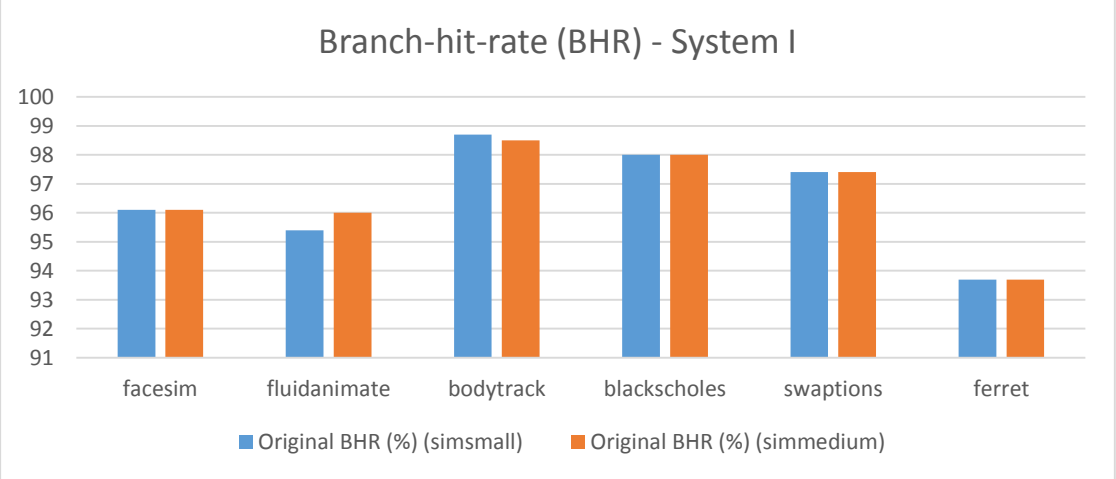
In the previous section we argue that performance characteristics of PARSEC applications do not change much with different inputs. In order to further test this hypothesis, we perform a different set of experiments. This time we compare IPC,



(a)



(b)



(c)

Figure 6.9. Performance characteristics of originals (PARSEC) running on simsmall and simmedium inputs on System-I showing (a) IPC, (b) CHR and (c) BHR values.

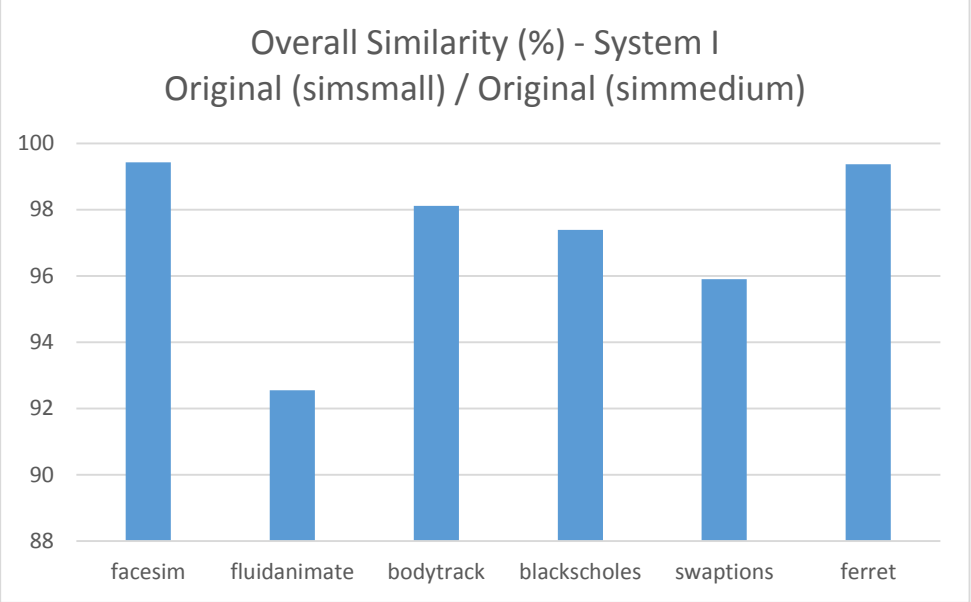


Figure 6.10. Overall similarities of originals (PARSEC) running on simsmall and simmedium inputs on System-I.

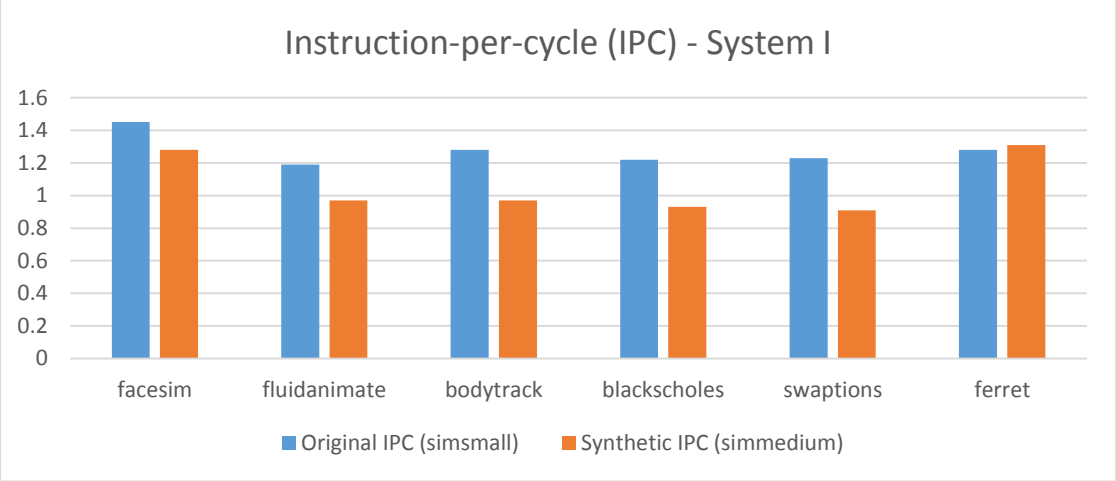
CMR and BMR values of originals running on simsmall input sets with the synthetics generated from originals running on simmedium input sets. This corresponds to the case where we ignore the inputs used in originals during synthesis and try to use a synthetic generated from an original application running on a particular input set as a proxy for the same original running on another input set.

Figure 6.11a shows IPC values of executions. Change in values have an average of 17% ranging from 2% for `ferret` to 26% for `swaptions`.

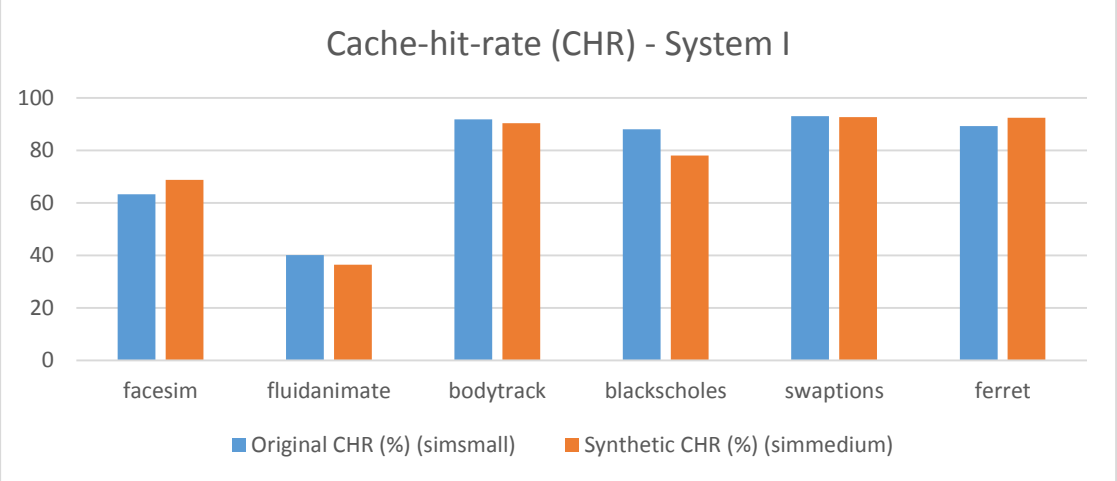
Figure 6.11b shows CHR values of executions. Change in values have an average of 5% ranging from 0.3% for `swaptions` to 8% for `facesim` and `fluidanimate`.

Figure 6.11c shows BHR values of executions. Change in values have an average of 0.5% ranging from 0% for `ferret` to 1.1% for `fluidanimate`.

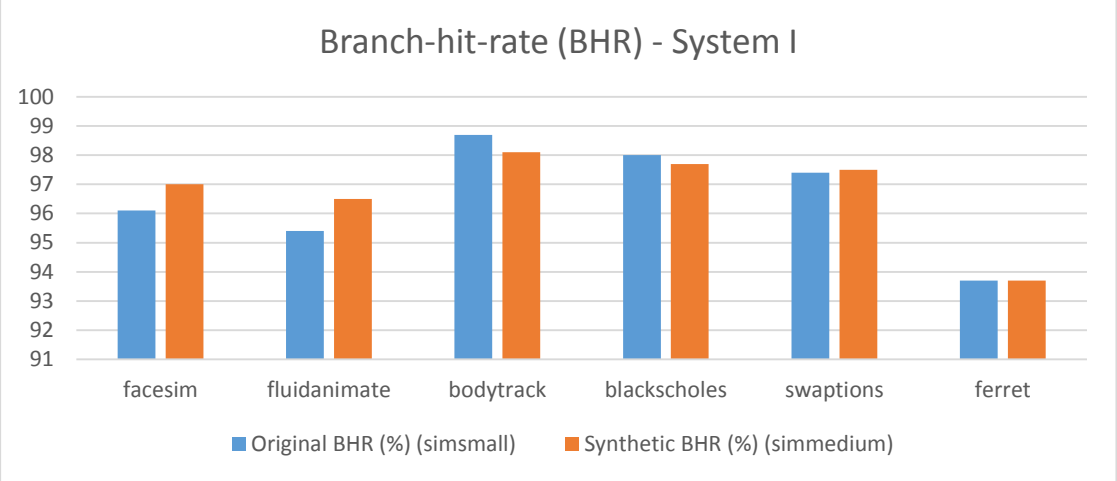
Figure 6.12 shows overall similarity of executions. Similarities have an average of 92% ranging from 88% for `blackscholes` to 98% for `ferret`.



(a)



(b)



(c)

Figure 6.11. Performance characteristics of originals (PARSEC) running on simsmall inputs and synthetics synthesized using simmedium inputs on System-I showing (a) IPC, (b) CHR and (c) BHR values.

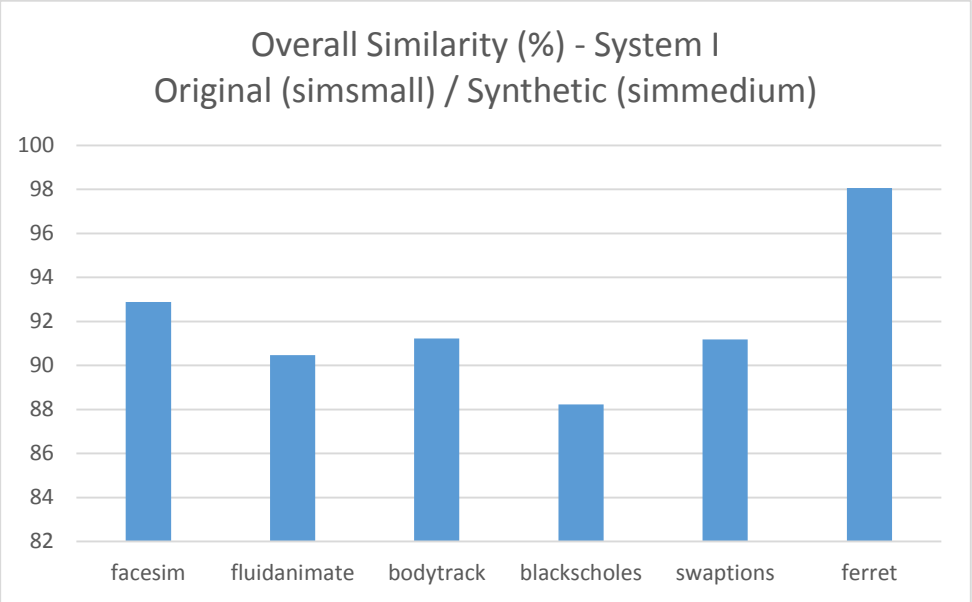


Figure 6.12. Overall similarities of originals (PARSEC) running on simsmall inputs and synthetics synthesized using simmedium inputs on System-I.

All changes lie in the acceptable error range used in our other experiments, that is 30% for individual and 20% for overall. This is in accordance with what we proposed earlier so that we confirm in this experiment that synthetics generated from PARSEC examples running on simmedium input set could be used as a proxy for same applications running on simsmall input set.

6.5. Speedups of Synthetics

In order to calculate speedups we compared execution times of original and synthetic applications. Average speedup is calculated not by averaging the speedups but dividing the total execution times of originals to synthetics.

Table 6.3 shows the speedups for PARSEC applications across different hardwares. The average speedup is 141 ranging from 44 for `ferret` to 230 for `swaptions` in the first hardware. For the second and third hardware the average speedup is similarly 145 and 129 respectively.

Table 6.3. Speedups of synthetics measured across different hardwares in P2S experiments.

Application	**Speedup (I)**	**Speedup (II)**	**Speedup (III)**
facesim	1115.6	1371.6	897.8
fluidanimate	29.2	33.6	31.6
bodytrack	59.7	61.9	54.2
blackscholes	47.5	44.3	48.0
swaptions	230.7	244.0	173.1
ferret	44.5	44.4	63.1
Average	**141.3**	**145.8**	**129.3**

Table 6.4 shows the speedups for SystemC applications across different hardwares. The average speedup is 4.2 ranging from 0.6 to 37 in the first hardware. 32 of 35 applications have a speedup above 1. Similarly for the second and third hardware the average speedup is 3.7 and 3.5 respectively.

Table 6.4. Speedups of synthetics measured across different hardwares in S2P experiments.

Application	**Speedup (I)**	**Speedup (II)**	**Speedup (III)**
tlm2_getting_started_1	2.6	2.4	1.8
tlm2_getting_started_4	37.5	25.5	20.6
transmitter	2.0	1.9	1.4
methods	2.5	2.5	2.5
simple_bus	14.9	17.6	6.1
gd-matmul	0.7	1.0	0.6
globals	2.9	3.0	2.4
ABS	9.5	5.9	4.3
gd-matmul3	0.7	0.8	0.9
toy	2.5	2.1	1.8

Table 6.4 – continued from previous page

scx_mutex_w_policy	2.0	1.9	3.0
sc_export	2.0	1.9	1.5
reset	3.6	2.2	3.1
doubleD	2.7	2.4	2.2
signal_versus_variable	3.1	2.5	2.9
tp-dummy2	2.0	1.9	1.9
kundu	2.0	1.6	0.5
call_backs	5.1	4.5	2.5
simple_fifo	2.8	2.4	1.6
multi_ports	2.5	2.5	1.8
dc-msort2	19.1	16.5	17.0
tlm2_getting_started_3	2.8	2.9	1.8
tp-dummy	1.8	1.9	3.5
pc-sfifo-1	1.9	1.6	1.2
combinational_versus ...	2.6	1.9	1.1
lt_extension_mandatory	2.9	2.1	1.4
dc-msort	8.0	7.8	7.6
tlm2_getting_started_2	3.0	2.3	2.4
control_flow_simple_if	2.6	2.7	1.7
dff	2.1	2.1	1.1
mem_slave5	1.4	1.6	0.6
partial_bus	3.1	1.8	1.9
specialized_signals	2.2	2.1	2.2
gd-matmul2	0.6	0.8	0.8
fir_trans_sym	4.8	1.6	2.4
Average	**4.2**	**3.7**	**3.5**

Results show that there is a huge gap between speedups for PARSEC and SystemC applications. The reason behind is that our set of SystemC examples mostly

Table 6.5. Convergence success percentages of SystemC applications using different individual/overall similarity thresholds.

	Converged	**Total**	**Success Percentage**
60%/70%	39	58	67.2%
70%/80%	35	58	60.3%
80%/90%	21	58	36.2%

consist of very simple designs and simulations finish almost instantly. Execution times of synthetics on the other hand are usually independent of the execution times of their originals. Hence the longer the execution time of the original application is the more speedup we get.

6.6. Convergence Success Percentages of SystemC Applications

We performed experiments with different similarity thresholds in SystemC examples to see how much it effects the convergence success percentage in our experiments. Our original set of examples consist of 75 applications in which 9 of them are single threaded. In another 8 applications, we have had problems in the instrumentation step. We excluded these applications from our experiments.

Table 6.5 shows the convergence success percentage for SystemC applications using different similarity thresholds. There seems to be a dramatic decrease at 80%/90% which drops the convergence success percentage below 50%. Hence we concluded that 70%/80% is a good compromise between accuracy and convergence success percentage and used these thresholds for the rest of our experiments.

6.7. Iteration Counts and Times

We have analysed number of iterations and total iteration times of applications in S2P experiments. Figure 6.13 shows the histogram plot of number of iterations.

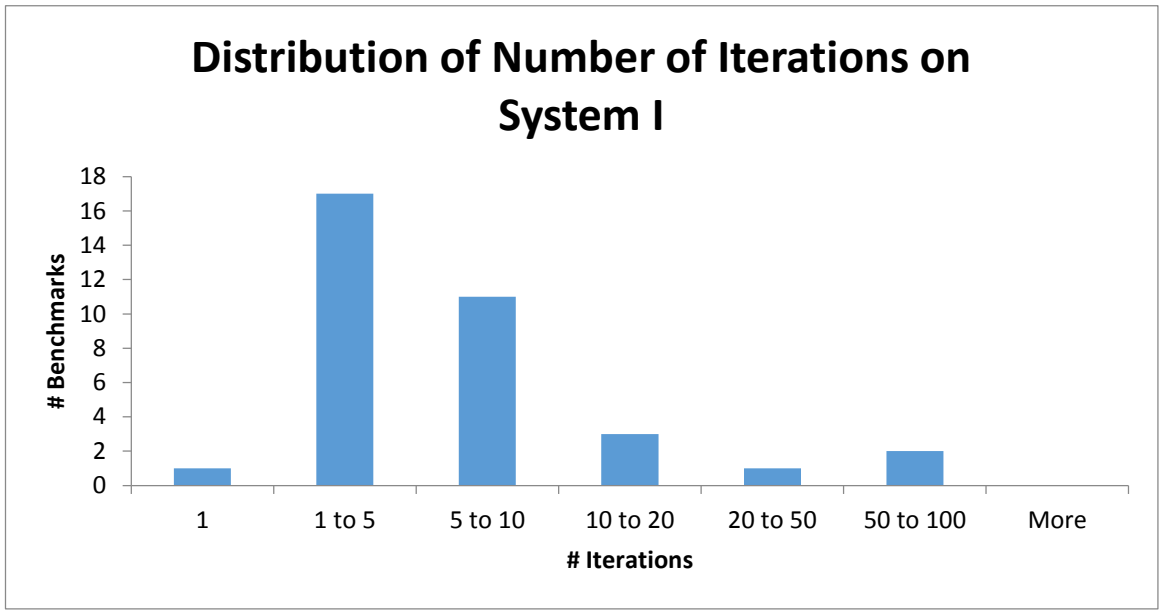


Figure 6.13. Distribution of number of iterations until convergence for SystemC applications.

Average number of iterations turned out to be 11.2 for our experiments.

Figure 6.14 shows the histogram plot of total iteration times. Average time it takes for a synthesis turned out to be 59.5 secs for our experiments.

One can say two figures given here are actually very similar to each other. They both reflect the very nature of our iterations, that is they either finish very quickly or last longer. Most frequently 1 to 5 iterations are enough for convergence and it goes up to 10 at other times. Synthesis usually takes under a minute if the instrumentation time of the original allows it to be.

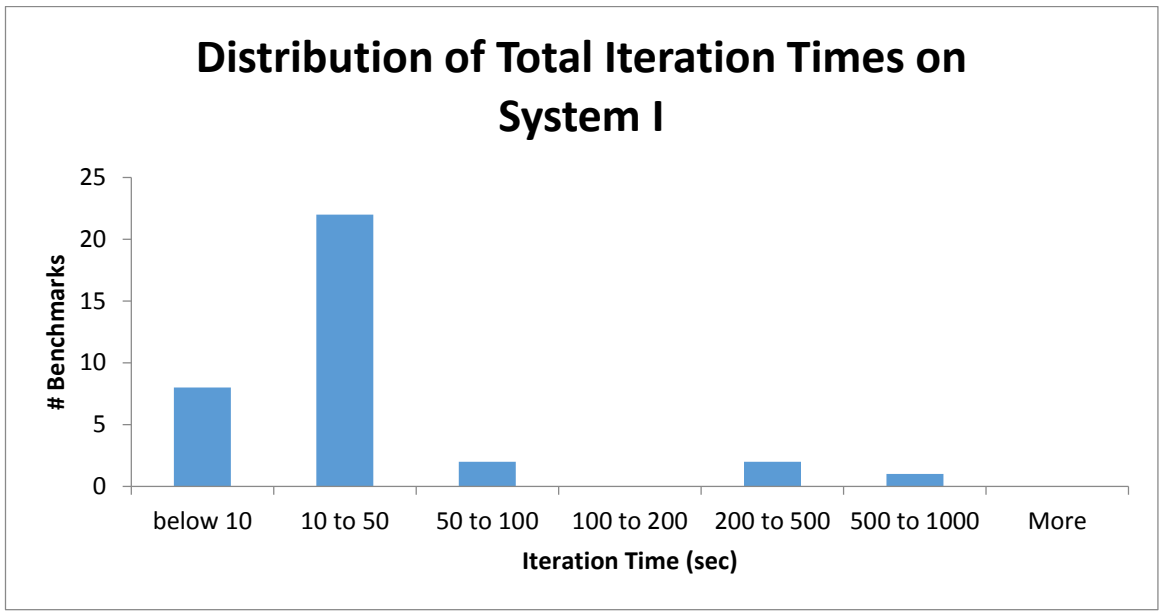


Figure 6.14. Distribution of total iteration times until convergence for SystemC applications.

7. CONCLUSIONS

We presented a framework that is extended from [1] to generate synthetic Pthreads benchmarks from SystemC/TLM applications (S2P) and SystemC benchmarks from Pthreads applications (P2S). We developed characterization techniques for SystemC based on high-level (parallel patterns) and low-level characteristics (IPC, BMR, and CMR). For similarity comparisons we only used the low-level characteristics which are related to performance.

Our experiments show that it is possible to create a synthetic program that has similar performance characteristics to the original one. Correctly matching the parallel pattern of the original benchmarks with the synthetic benchmarks enables portability of our solutions on different hardwares. Experiments on three different hardwares show that mimicking performance similarities with acceptable error percentages are quite possible across different systems.

Furthermore, the synthetic runs faster leading to speeding up simulations for architectural exploration. Average speedup of our examples are 141x for Pthreads applications and 4x for SystemC applications. If one uses long running SystemC applications as well, synthetics would yield much higher speedup values.

7.1. Limitations

Our approach still has several limitations. First of all, we can not guarantee portability on different architectures such as PowerPC since our low-level code blocks may perform differently on these machines. This problem also arises when systems have the same architecture but quite different specifications. For example code blocks we used for CMR assume a maximum cache size so that we get arbitrary results on machines with caches larger than this value. We believe this is the main reason for the portability problems we had in the third system we used in our experiments.

Furthermore, Our synthetics do not have the exact same synchronization structure of their originals since this is not a translator work. What we have is the same number of threads as the originals and to some extend the same threading structure as much as captured with our high-level metrics. For this reason, we think our approach is not usable in compiler studies.

Lastly, the examples we used in our experiments are assumed to have only a single parallel pattern whereas real world applications often make use of multiple patterns in a single application. This is mostly a limitation for not having any semantics in the binary of the application so that it becomes very difficult or even impossible to detect different patterns used in different parts of the application.

7.2. Future Work

Possible directions for future work include increasing the similarity of synthetics by improving both convergence success percentage and the portability. It is possible to divide the applications into different phases and then try to mimic each phase individually. In order to do this one needs to find and use larger benchmarks as inputs for both Pthreads and SystemC applications. This might require an optimization in the instrumentation algorithms used when analysing the original applications.

Other than that there are some studies which tries to estimate power usages in SystemC and TLM models [38, 39, 40]. We think having synthetics mimicking the same power usage characteristics of their originals may have some applications in the industry.

Lastly, it may be of use to implement a standalone benchmark synthesizer that would output a synthetic based on the characteristics given as input. This should eliminate the need to have an original application in the first place in order to get a synthetics version with desired characteristics.

APPENDIX A: PINTOOLS

A.1. tname_pthread.cpp

```
// ============================================================================ //
//
// tname_pthread -- Extract names of threads in a pthread application
//
// Last modified: Mon Feb 17, 2014  03:55PM
// Authors:       Gokcehan Kara <gokcehan.kara@boun.edu.tr>
//                Alper Sen     <alper.sen@boun.edu.tr>
//
// ============================================================================ //

#include <stdio.h>
#include "pin.H"

// ============================================================================ //
//  Analysis Routines
// ============================================================================ //
// Note:  threadid+1 is used as an argument to the GetLock()
//        routine as a debugging aid.  This is the value that
//        the lock is set to, so it must be non-zero.

// lock serializes access to the output file.
FILE * tnames_pthread;
PIN_LOCK lock;

// This routine is executed each time malloc is called.
VOID BeforePThreadCreate( int size, THREADID threadid )
{
    GetLock(&lock, threadid+1);
    PIN_LockClient();
    string rtn_name = RTN_Name(RTN_FindByAddress(size));
    PIN_UnlockClient();
    fprintf(tnames_pthread, "%s\n", rtn_name.c_str());
    fflush(tnames_pthread);
    ReleaseLock(&lock);
}


// ============================================================================ //
// Instrumentation Routines
// ============================================================================ //

// This routine is executed for each image.
```

```
VOID ImageLoad(IMG img, VOID *)
{
    RTN rtn = RTN_FindByName(img, "pthread_create");
    if (RTN_Valid(rtn)) {
        RTN_Open(rtn);
        RTN_InsertCall(rtn, IPOINT_BEFORE, AFUNPTR(BeforePThreadCreate),
                       IARG_FUNCARG_ENTRYPOINT_VALUE, 2,
                       IARG_THREAD_ID, IARG_END);
        RTN_Close(rtn);
    }
}

// This routine is executed once at the end.
VOID Fini(INT32 code, VOID *v)
{
    fprintf(tnames_pthread, "main\n");
    fclose(tnames_pthread);
}

// ================================================================== //
// Main                                                               //
// ================================================================== //

int main(INT32 argc, CHAR **argv)
{
    // Initialize the pin lock
    InitLock(&lock);

    // Initialize pin
    PIN_Init(argc, argv);
    PIN_InitSymbols();

    tnames_pthread = fopen("tnames_pthread.out", "w");

    // Register ImageLoad to be called when each image is loaded.
    IMG_AddInstrumentFunction(ImageLoad, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Never returns
    PIN_StartProgram();

    return 0;
}
```

A.2. tname_systemc.cpp

```
// ============================================================================ //
//
// tname_systemc -- Extract names of threads in a systemc application
//
// Last modified: Mon Feb 17, 2014  03:56PM
// Authors:       Gokcehan Kara <gokcehan.kara@boun.edu.tr>
//                Alper Sen     <alper.sen@boun.edu.tr>
//
// ============================================================================ //

#include <iostream>
#include <fstream>
#include "pin.H"

PIN_LOCK lock;
static ofstream thread_names_out;

VOID ThreadBefore(CHAR * name)
{
    thread_names_out << name << endl;
}

VOID Image(IMG img, VOID *v)
{
    for (SYM sym = IMG_RegsymHead(img); SYM_Valid(sym); sym = SYM_Next(sym)) {
        std::string funcName = PIN_UndecorateSymbolName(SYM_Name(sym),
                                UNDECORATION_COMPLETE);

        string creator = "sc_core::sc_simcontext::create_thread_process";
        if (std::string::npos != funcName.find(creator)) {
            RTN funcCall = RTN_FindByName(img, SYM_Name(sym).c_str());
            if (RTN_Valid(funcCall)) {
                RTN_Open(funcCall);
                RTN_InsertCall(funcCall, IPOINT_BEFORE, (AFUNPTR)ThreadBefore,
                                IARG_FUNCARG_CALLSITE_VALUE, 2,
                                IARG_END);
                RTN_Close(funcCall);
            }
        }
    }
}

// ============================================================================ //
// Main                                                                         //
// ============================================================================ //

// This routine is executed once at the end.
```

```
VOID Fini(INT32 code, VOID *v)
{
    thread_names_out << "sc_main" << endl;
}

int main(int argc, char *argv[])
{
    PIN_InitSymbols();
    PIN_Init(argc, argv);

    thread_names_out.open("tnames_systemc.out");
    if (thread_names_out.is_open()) {
        cout << "creating \"tnames_systemc.out\"..." << endl;
    } else {
        cout << "\"tnames_systemc.out\" cannot be created." << endl;
    }

    IMG_AddInstrumentFunction(Image, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();

    return 0;
}
```

A.3. shadow_pthread.cpp

```
// ============================================================================ //
//
// shadow_pthread -- Analyse a pthread program to record statistics
//
// Last modified: Mon Feb 17, 2014  03:51PM
// Authors:       Gokcehan Kara <gokcehan.kara@boun.edu.tr>
//                Alper Sen     <alper.sen@boun.edu.tr>
//
// ============================================================================ //
//
// sample output file (pin.a.log) looks as follows:
//
// PC of Thread 2: 4195952
// Creator of Thread 2: 1
// ..
// Dependency of Thread 1: [0] 0, [2] 0, [3] 0, [4] 0, [5] 0, [6] 0
// Correlation of Thread 1: [0] 0, [2] 0, [3] 0, [4] 0, [5] 0, [6] 0
// ..
// Instruction Count of Thread 1: 315
// ..
// Total Load: 123
// Total Store: 79
// Total Instruction Count of Process: 360
// total private of group 1: 23
// total readonly of group 1: 18
// total producer/consumer of group 1: 2
// total migratory of group 1: 16
// total cacheline of group 1: 59
// Time of Thread 1: 13.49.37.47 - 13.49.37.177
// ..
//

#include <stdio.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <bitset>
#include <unordered_map>  // c++11
#include <sys/time.h>     // linux
#include "pin.H"

PIN_LOCK lock;
INT32 numThreads = 0;

static ofstream trace_file("pin.a.log");

static ifstream thread_names_inp;
```

```
static vector<string> thread_names;

static const char max_num_threads = 32;  // arbitrary

static long private_var_count;
static long readonly_var_count;
static long writeonly_var_count;
static long prodcons_var_count;
static long migratory_var_count;
static long total_var_count;
static long other_var_count;

static long read_instruction_count;
static long write_instruction_count;
static long total_instruction_count;
static long instruction_counts_of_threads[max_num_threads];

static long dependency_matrix[max_num_threads][max_num_threads];

class thread_data_t {
public:
    timeval thread_start_time;
    timeval thread_finish_time;
    ADDRINT PC_of_thread;
    THREADID creator_of_thread;
    OS_THREAD_ID otid;
    THREADID tid;
    UINT64 _count;
    thread_data_t() : PC_of_thread(-1),
                      creator_of_thread(-1),
                      otid(-1),
                      tid(-1),
                      _count(0) {}
};

// key for accessing TLS storage in the threads. initialized once in main()
static TLS_KEY tls_key;

// function to access thread-specific data
thread_data_t* get_tls(THREADID threadid)
{
    thread_data_t* tdata =
            static_cast<thread_data_t*>(PIN_GetThreadData(tls_key, threadid));
    return tdata;
}

struct shadow_mem {
    char last_reader;
```

```
    char last_writer;
    bitset<max_num_threads> readers;
    bitset<max_num_threads> writers;
    shadow_mem() {
        last_reader = -1;
        last_writer = -1;
    }
};

static unordered_map<size_t, shadow_mem> shadow_mem_table;

// Print a memory read record
VOID RecordMemRead(VOID * ip, VOID * addr, THREADID tid)
{
    GetLock(&lock, tid+1);

    read_instruction_count++;

    unordered_map<size_t, shadow_mem>::iterator it;
    it = shadow_mem_table.find((size_t)addr);
    if (it != shadow_mem_table.end()) {
        int last = shadow_mem_table[(size_t)addr].last_writer;
        if (last != -1) dependency_matrix[tid][last]++;
        shadow_mem_table[(size_t)addr].last_reader = tid;
        shadow_mem_table[(size_t)addr].readers[tid] = true;
    } else {
        shadow_mem s;
        s.last_reader = tid;
        s.readers[tid] = true;
        shadow_mem_table[(size_t)addr] = s;
    }

    ReleaseLock(&lock);
}

// Print a memory write record
VOID RecordMemWrite(VOID * ip, VOID * addr, THREADID tid)
{
    GetLock(&lock, tid+1);

    write_instruction_count++;

    unordered_map<size_t, shadow_mem>::iterator it;
    it = shadow_mem_table.find((size_t)addr);
    if (it != shadow_mem_table.end()) {
        shadow_mem_table[(size_t)addr].last_writer = tid;
        shadow_mem_table[(size_t)addr].writers[tid] = true;
    } else {
```

```
            shadow_mem s;
            s.last_writer = tid;
            s.writers[tid] = true;
            shadow_mem_table[(size_t)addr] = s;
        }

        ReleaseLock(&lock);
}

VOID inscount(THREADID tid)
{
        GetLock(&lock, tid+1);
        total_instruction_count++;
        instruction_counts_of_threads[tid]++;
        ReleaseLock(&lock);

        thread_data_t* tdata = get_tls(tid);
        tdata->_count++;
}

// Is called for every instruction and instruments reads and writes
VOID Instruction(INS ins, VOID *v)
{
        // filter out library calls
        IMG img = IMG_FindByAddress(INS_Address(ins));
        if (!IMG_Valid(img)) return;
        if (!IMG_IsMainExecutable(img)) return;

        // regular instruction counting
        INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR)inscount,
                IARG_THREAD_ID,
                IARG_END);

        // Instruments memory accesses using a predicated call, i.e.
        // the instrumentation is called iff the instruction will actually be
        // executed.
        //
        // The IA-64 architecture has explicitly predicated instructions.
        // On the IA-32 and Intel(R) 64 architectures conditional moves and REP
        // prefixed instructions appear as predicated instructions in Pin.
        UINT32 memOperands = INS_MemoryOperandCount(ins);

        // Iterate over each memory operand of the instruction.
        for (UINT32 memOp = 0; memOp < memOperands; memOp++) {
            if (INS_MemoryOperandIsRead(ins, memOp)) {
                INS_InsertPredicatedCall(
                        ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
```

```
                    IARG_INST_PTR,
                    IARG_MEMORYOP_EA, memOp,
                    IARG_THREAD_ID,
                    IARG_END);
        }
        // Note that in some architectures a single memory operand can be
        // both read and written (for instance incl (%eax) on IA-32)
        // In that case we instrument it once for read and once for write.
        if (INS_MemoryOperandIsWritten(ins, memOp)) {
            INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE, (AFUNPTR)RecordMemWrite,
                    IARG_INST_PTR,
                    IARG_MEMORYOP_EA, memOp,
                    IARG_THREAD_ID,
                    IARG_END);
        }
    }
}

void extract_variable_types()
{
    unordered_map<size_t, shadow_mem>::iterator it;
    for (it = shadow_mem_table.begin(); it != shadow_mem_table.end(); ++it) {
        int cons = 0;
        for (size_t i = 0; i < it->second.readers.size(); ++i) {
            if (it->second.readers[i] == 1) cons++;
        }
        int prod = 0;
        for (size_t i = 0; i < it->second.writers.size(); ++i) {
            if (it->second.writers[i] == 1) prod++;
        }

        if ((cons == 1 && prod == 1) &&
            (it->second.last_reader == it->second.last_writer)) {
            private_var_count++;
        } else if (prod == 0) {  // (cons == 0) should not happen
            readonly_var_count++;
        } else if (cons > prod) {
            prodcons_var_count++;
        } else if (cons <= prod) {
            if (prod == 0) {
                writeonly_var_count++;
            } else {
                migratory_var_count++;
            }
        } else {
            other_var_count++;
        }
```

```
        total_var_count++;
    }
}

// This routine is executed every time a thread is started.
VOID ThreadStart(THREADID threadid, CONTEXT *ctxt, INT32 flags, VOID *v)
{
    GetLock(&lock, threadid+1);
    numThreads++;
    ReleaseLock(&lock);

    thread_data_t* tdata = new thread_data_t;

    PIN_SetThreadData(tls_key, tdata, threadid);
}

// This routine is executed every time a thread is destroyed.
VOID ThreadFini(THREADID threadid, const CONTEXT *ctxt, INT32 code, VOID *v)
{
    GetLock(&lock, threadid+1);
    thread_data_t* tdata = get_tls(threadid);
    gettimeofday(&tdata->thread_finish_time, NULL);
    ReleaseLock(&lock);
}

VOID BeforePthread(ADDRINT addr, THREADID tid)
{
    thread_data_t* tdata = get_tls(tid);
    gettimeofday(&tdata->thread_start_time, NULL);
    tdata->PC_of_thread = addr;
    tdata->otid = PIN_GetTid();
    tdata->tid = PIN_ThreadId();
    tdata->creator_of_thread = PIN_GetParentTid();
}

VOID Image(IMG img, VOID *v)
{
    vector<string>::iterator it;
    for (it = thread_names.begin(); it != thread_names.end(); ++it) {
        RTN rtn = RTN_FindByName(img, it->c_str());
        if (RTN_Valid(rtn)) {
            RTN_Open(rtn);
            ADDRINT addr = INS_Address(RTN_InsHead(rtn));
            RTN_InsertCall(rtn, IPOINT_BEFORE, AFUNPTR(BeforePthread),
                           IARG_ADDRINT, addr,
                           IARG_THREAD_ID, IARG_END);
            RTN_Close(rtn);
        }
```

```
    }
}

VOID Fini(INT32 code, VOID *v)
{
    // PC and Creator of Threads -- dummy fields for the generator
    unordered_map<int, int> m;
    for (INT32 t=0; t<numThreads; t++) {
        thread_data_t* tdata = get_tls(t);
        m[tdata->otid] = tdata->tid;
    }
    for (INT32 t=1; t<numThreads; t++) {
        thread_data_t* tdata = get_tls(t);
        trace_file << "PC of Thread " << tdata->tid + 1 << ": "
                   << tdata->PC_of_thread << endl;
        trace_file << "Creator of Thread " << tdata->tid + 1 << ": "
                   << m[tdata->creator_of_thread] + 1 << endl;
    }

    // Dependency and Correlation of Threads -- same values for both
    for (INT32 i = 0; i < numThreads; ++i) {
        trace_file << "Dependency of Thread " << i + 1 << ": ";
        trace_file << "[0] 0";  // dummy
        for (INT32 j = 0; j < numThreads; ++j) {
            if (i == j) continue;
            trace_file << ", [" << j + 1 << "] " << dependency_matrix[i][j];
        }
        trace_file << endl;

        trace_file << "Correlation of Thread " << i + 1 << ": ";
        trace_file << "[0] 0";  // dummy
        for (INT32 j = 0; j < numThreads; ++j) {
            if (i == j) continue;
            trace_file << ", [" << j + 1 << "] " << dependency_matrix[i][j];
        }
        trace_file << endl;
    }

    // Instruction Count of Threads
    for (INT32 t=0; t<numThreads; t++) {
        thread_data_t* tdata = get_tls(t);
        trace_file << "Instruction Count of Thread " << t + 1 << ": "
                   << tdata->_count << endl;
    }

    // Load/Store/Total Instructions
    trace_file << "Total Load: " << read_instruction_count << endl;
    trace_file << "Total Store: " << write_instruction_count << endl;
```

```
    trace_file << "Total Instruction Count of Process: "
               << total_instruction_count << endl;

    // private, producer/consumer, readonly, migratory variables
    extract_variable_types();
    trace_file << "total private of group 1: "
               << private_var_count << endl;
    trace_file << "total readonly of group 1: "
               << readonly_var_count << endl;
    // trace_file << "total writeonly of group 1: "
    //            << writeonly_var_count << endl;
    trace_file << "total producer/consumer of group 1: "
               << prodcons_var_count << endl;
    trace_file << "total migratory of group 1: "
               << migratory_var_count << endl;
    // trace_file << "other private of group 1: "
    //            << other_var_count << endl;
    trace_file << "total cacheline of group 1: "
               << total_var_count << endl;

    // Time of Threads
    for (INT32 i=0; i<numThreads; i++) {
        thread_data_t* tdata = get_tls(i);
        trace_file << "Time of Thread " << i + 1 << ": ";

        long t;
        int msec, sec, min, hour;

        t = 1000 * tdata->thread_start_time.tv_sec
            + tdata->thread_start_time.tv_usec / 1000;  // milisecond

        msec = t % 1000;
        sec  = (t / 1000) % 60;
        min  = ((t / 1000) / 60) % 60;
        hour = (((t / 1000) / 60) / 60) % 24;
        trace_file << hour << '.' << min << '.' << sec << '.' << msec << " - ";

        t = 1000 * tdata->thread_finish_time.tv_sec
            + tdata->thread_finish_time.tv_usec / 1000;  // milisecond

        msec = t % 1000;
        sec  = (t / 1000) % 60;
        min  = ((t / 1000) / 60) % 60;
        hour = (((t / 1000) / 60) / 60) % 24;
        trace_file << hour << '.' << min << '.' << sec << '.' << msec << endl;
    }
}
```

```
// ================================================================== //
// Main                                                               //
// ================================================================== //

int main(int argc, char *argv[])
{
    PIN_InitSymbols();
    PIN_Init(argc, argv);

    thread_names_inp.open("tnames_pthread.out");
    if (thread_names_inp.is_open()) {
        cout << "\"tnames_pthread.out\" is loading.." << endl;
        string name;
        while (thread_names_inp >> name) thread_names.push_back(name);
        thread_names_inp.close();
    } else {
        cout << "\"tnames_pthread.out\" cannot be loaded." << endl;
    }

    // Initialize the lock
    InitLock(&lock);

    // Obtain  a key for TLS storage.
    tls_key = PIN_CreateThreadDataKey(0);

    INS_AddInstrumentFunction(Instruction, 0);
    IMG_AddInstrumentFunction(Image, 0);
    PIN_AddFiniFunction(Fini, 0);

    PIN_AddThreadStartFunction(ThreadStart, 0);
    PIN_AddThreadFiniFunction(ThreadFini, 0);

    // Never returns
    PIN_StartProgram();

    return 0;
}
```

A.4. shadow_systemc.cpp

```
// ============================================================================ //
//
// shadow_systemc -- Analyse a systemc program to record statistics
//
// Last modified: Mon Feb 17, 2014  05:21PM
// Authors:       Gokcehan Kara <gokcehan.kara@boun.edu.tr>
//                Alper Sen     <alper.sen@boun.edu.tr>
//
// ============================================================================ //
//
// sample output file (pin.a.log) looks as follows:
//
// PC of Thread 2: 4201874
// Creator of Thread 2: 1
// ..
// Dependency of Thread 1: [0] 0, [2] 0
// Correlation of Thread 1: [0] 0, [2] 0
// ..
// Instruction Count of Thread 1: 660
// ..
// Total Load: 297
// Total Store: 160
// Total Instruction Count of Process: 670
// total private of group 1: 35
// total readonly of group 1: 33
// total producer/consumer of group 1: 0
// total migratory of group 1: 27
// total cacheline of group 1: 95
// Time of Thread 1: 13.52.36.778 - 13.52.37.976
// ..
//

#include <stdio.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <bitset>
#include <unordered_map>  // c++11
#include <sys/time.h>     // linux
#include "pin.H"

PIN_LOCK lock;
INT32 numThreads = 0;

static ofstream trace_file("pin.a.log");

static ifstream thread_names_inp;
```

```
static vector<string> thread_names;

static const char max_num_threads = 32;  // arbitrary

static long private_var_count;
static long readonly_var_count;
static long writeonly_var_count;
static long prodcons_var_count;
static long migratory_var_count;
static long total_var_count;
static long other_var_count;

static long read_instruction_count;
static long write_instruction_count;
static long total_instruction_count;

static long dependency_matrix[max_num_threads][max_num_threads];

class thread_data_t {
public:
    timeval thread_start_time;
    timeval thread_finish_time;
    ADDRINT PC_of_thread;
    THREADID creator_of_thread;
    OS_THREAD_ID otid;
    THREADID tid;
    UINT64 _count;
    thread_data_t() : PC_of_thread(0),
                      creator_of_thread(-1),
                      otid(-1),
                      tid(-1),
                      _count(0)
    {
        thread_start_time.tv_sec = 0;
        thread_start_time.tv_usec = 0;
    }
};

// key for accessing TLS storage in the threads. initialized once in main()
static TLS_KEY tls_key;

// function to access thread-specific data
thread_data_t* get_tls(THREADID threadid)
{
    thread_data_t* tdata =
            static_cast<thread_data_t*>(PIN_GetThreadData(tls_key, threadid));
    return tdata;
}
```

```
struct shadow_mem {
    char last_reader;
    char last_writer;
    bitset<max_num_threads> readers;
    bitset<max_num_threads> writers;
    shadow_mem() {
        last_reader = -1;
        last_writer = -1;
    }
};

static unordered_map<size_t, shadow_mem> shadow_mem_table;

// Print a memory read record
VOID RecordMemRead(VOID * ip, VOID * addr, THREADID tid)
{
    GetLock(&lock, tid+1);

    read_instruction_count++;

    unordered_map<size_t, shadow_mem>::iterator it;
    it = shadow_mem_table.find((size_t)addr);
    if (it != shadow_mem_table.end()) {
        int last = shadow_mem_table[(size_t)addr].last_writer;
        if (last != -1) dependency_matrix[tid][last]++;
        shadow_mem_table[(size_t)addr].last_reader = tid;
        shadow_mem_table[(size_t)addr].readers[tid] = true;
    } else {
        shadow_mem s;
        s.last_reader = tid;
        s.readers[tid] = true;
        shadow_mem_table[(size_t)addr] = s;
    }

    ReleaseLock(&lock);
}

// Print a memory write record
VOID RecordMemWrite(VOID * ip, VOID * addr, THREADID tid)
{
    GetLock(&lock, tid+1);

    write_instruction_count++;

    unordered_map<size_t, shadow_mem>::iterator it;
    it = shadow_mem_table.find((size_t)addr);
    if (it != shadow_mem_table.end()) {
```

```
            shadow_mem_table[(size_t)addr].last_writer = tid;
            shadow_mem_table[(size_t)addr].writers[tid] = true;
        } else {
            shadow_mem s;
            s.last_writer = tid;
            s.writers[tid] = true;
            shadow_mem_table[(size_t)addr] = s;
        }

        ReleaseLock(&lock);
}

VOID inscount(THREADID tid)
{
        GetLock(&lock, tid+1);
        total_instruction_count++;
        ReleaseLock(&lock);

        thread_data_t* tdata = get_tls(tid);
        tdata->_count++;
}

// Is called for every instruction and instruments reads and writes
VOID Instruction(INS ins, VOID *v)
{
        // filter out library calls
        IMG img = IMG_FindByAddress(INS_Address(ins));
        if (!IMG_Valid(img)) return;
        if (!IMG_IsMainExecutable(img)) return;

        // regular instruction counting
        INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR)inscount,
                IARG_THREAD_ID,
                IARG_END);

        // Instruments memory accesses using a predicated call, i.e.
        // the instrumentation is called iff the instruction will actually be
        // executed.
        //
        // The IA-64 architecture has explicitly predicated instructions.
        // On the IA-32 and Intel(R) 64 architectures conditional moves and REP
        // prefixed instructions appear as predicated instructions in Pin.
        UINT32 memOperands = INS_MemoryOperandCount(ins);

        // Iterate over each memory operand of the instruction.
        for (UINT32 memOp = 0; memOp < memOperands; memOp++) {
            if (INS_MemoryOperandIsRead(ins, memOp)) {
```

```
            INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
                    IARG_INST_PTR,
                    IARG_MEMORYOP_EA, memOp,
                    IARG_THREAD_ID,
                    IARG_END);
        }
        // Note that in some architectures a single memory operand can be
        // both read and written (for instance incl (%eax) on IA-32)
        // In that case we instrument it once for read and once for write.
        if (INS_MemoryOperandIsWritten(ins, memOp)) {
            INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE, (AFUNPTR)RecordMemWrite,
                    IARG_INST_PTR,
                    IARG_MEMORYOP_EA, memOp,
                    IARG_THREAD_ID,
                    IARG_END);
        }
    }
}

void extract_variable_types()
{
    unordered_map<size_t, shadow_mem>::iterator it;
    for (it = shadow_mem_table.begin(); it != shadow_mem_table.end(); ++it) {
        int cons = 0;
        for (size_t i = 0; i < it->second.readers.size(); ++i) {
            if (it->second.readers[i] == 1) cons++;
        }
        int prod = 0;
        for (size_t i = 0; i < it->second.writers.size(); ++i) {
            if (it->second.writers[i] == 1) prod++;
        }

        if ((cons == 1 && prod == 1) &&
            (it->second.last_reader == it->second.last_writer)) {
            private_var_count++;
        } else if (prod == 0) {  // (cons == 0) should not happen
            readonly_var_count++;
        } else if (cons > prod) {
            prodcons_var_count++;
        } else if (cons <= prod) {
            if (prod == 0) {
                writeonly_var_count++;
            } else {
                migratory_var_count++;
            }
        } else {
```

```
            other_var_count++;
        }
        total_var_count++;
    }
}

// This routine is executed every time a thread is started.
VOID ThreadStart(THREADID threadid, CONTEXT *ctxt, INT32 flags, VOID *v)
{
    GetLock(&lock, threadid+1);
    numThreads++;
    ReleaseLock(&lock);

    thread_data_t* tdata = new thread_data_t;

    PIN_SetThreadData(tls_key, tdata, threadid);
    if (threadid == 0) {
        gettimeofday(&tdata->thread_start_time, NULL);
    }
}

// This routine is executed every time a thread is destroyed.
VOID ThreadFini(THREADID threadid, const CONTEXT *ctxt, INT32 code, VOID *v)
{
    GetLock(&lock, threadid+1);
    thread_data_t* tdata = get_tls(threadid);
    gettimeofday(&tdata->thread_finish_time, NULL);
    ReleaseLock(&lock);
}

VOID BeforeSCThread(ADDRINT addr, THREADID tid) {
    PIN_LockClient();
    PIN_UnlockClient();
    thread_data_t* tdata = get_tls(tid);
    gettimeofday(&tdata->thread_start_time, NULL);
    tdata->PC_of_thread = addr;
    tdata->otid = PIN_GetTid();
    tdata->tid = PIN_ThreadId();
    tdata->creator_of_thread = PIN_GetParentTid();
}

VOID Image(IMG img, VOID *v)
{
    vector<string>::iterator it;
    for (it = thread_names.begin(); it != thread_names.end(); ++it) {
        string tname = *it;

        if (it->find("b2nb_thread") != string::npos) { tname = "b2nb_thread"; }
```

```
        if (it->find("nb2b_thread") != string::npos) { tname = "nb2b_thread"; }

        for (SYM sym = IMG_RegsymHead(img); SYM_Valid(sym); sym = SYM_Next(sym)) {
            string funcName = PIN_UndecorateSymbolName(SYM_Name(sym),
                                                       UNDECORATION_COMPLETE);
            if (string::npos != funcName.find(tname + '(')) {
                RTN funcCall = RTN_FindByName(img, SYM_Name(sym).c_str());
                if (RTN_Valid(funcCall)) {
                    RTN_Open(funcCall);
                    ADDRINT addr = INS_Address(RTN_InsHead(funcCall));
                    RTN_InsertCall(funcCall, IPOINT_BEFORE, AFUNPTR(BeforeSCThread),
                                   IARG_ADDRINT, addr,
                                   IARG_THREAD_ID, IARG_END);
                    RTN_Close(funcCall);
                }
            }
        }
    }
}

VOID Fini(INT32 code, VOID *v)
{
    // PC and Creator of Threads
    unordered_map<int, int> m;
    for (INT32 t=0; t<numThreads; t++) {
        thread_data_t* tdata = get_tls(t);
        m[tdata->otid] = tdata->tid;
    }
    for (INT32 t=1; t<numThreads; t++) {
        thread_data_t* tdata = get_tls(t);
        trace_file << "PC of Thread " << tdata->tid + 1 << ": "
                   << tdata->PC_of_thread << endl;
        trace_file << "Creator of Thread " << tdata->tid + 1 << ": "
                   << m[tdata->creator_of_thread] + 1 << endl;
    }

    // Dependency and Correlation of Threads -- same values for both
    for (INT32 i = 0; i < numThreads; ++i) {
        trace_file << "Dependency of Thread " << i + 1 << ": ";
        trace_file << "[0] 0";  // dummy
        for (INT32 j = 0; j < numThreads; ++j) {
            if (i == j) continue;
            trace_file << ", [" << j + 1 << "] " << dependency_matrix[i][j];
        }
        trace_file << endl;

        trace_file << "Correlation of Thread " << i + 1 << ": ";
        trace_file << "[0] 0";  // dummy
```

```
    for (INT32 j = 0; j < numThreads; ++j) {
        if (i == j) continue;
        trace_file << ", [" << j + 1 << "] " << dependency_matrix[i][j];
    }
    trace_file << endl;
}

// Instruction Count of Threads
for (INT32 t=0; t<numThreads; t++)
{
    thread_data_t* tdata = get_tls(t);
    trace_file << "Instruction Count of Thread " << t + 1 << ": "
               << tdata->_count << endl;
}

// Load/Store/Total Instructions
trace_file << "Total Load: " << read_instruction_count << endl;
trace_file << "Total Store: " << write_instruction_count << endl;
trace_file << "Total Instruction Count of Process: "
           << total_instruction_count << endl;

// private, producer/consumer, readonly, migratory variables
extract_variable_types();
trace_file << "total private of group 1: "
           << private_var_count << endl;
trace_file << "total readonly of group 1: "
           << readonly_var_count << endl;
// trace_file << "total writeonly of group 1: "
//            << writeonly_var_count << endl;
trace_file << "total producer/consumer of group 1: "
           << prodcons_var_count << endl;
trace_file << "total migratory of group 1: "
           << migratory_var_count << endl;
// trace_file << "other private of group 1: "
//            << other_var_count << endl;
trace_file << "total cacheline of group 1: "
           << total_var_count << endl;

// Time of Threads
for (INT32 i=0; i<numThreads; i++) {
    thread_data_t* tdata = get_tls(i);
    trace_file << "Time of Thread " << i + 1 << ": ";

    long t;
    int msec, sec, min, hour;

    t = 1000 * tdata->thread_start_time.tv_sec
        + tdata->thread_start_time.tv_usec / 1000;  // milisecond
```

```
        msec = t % 1000;
        sec  = (t / 1000) % 60;
        min  = ((t / 1000) / 60) % 60;
        hour = (((t / 1000) / 60) / 60) % 24;
        trace_file << hour << '.' << min << '.' << sec << '.' << msec << " - ";

        t = 1000 * tdata->thread_finish_time.tv_sec
            + tdata->thread_finish_time.tv_usec / 1000;  // milisecond

        msec = t % 1000;
        sec  = (t / 1000) % 60;
        min  = ((t / 1000) / 60) % 60;
        hour = (((t / 1000) / 60) / 60) % 24;
        trace_file << hour << '.' << min << '.' << sec << '.' << msec << endl;
    }
}

// ================================================================== //
// Main                                                               //
// ================================================================== //

int main(int argc, char *argv[])
{
    PIN_InitSymbols();
    PIN_Init(argc, argv);

    thread_names_inp.open("tnames_systemc.out");
    if (thread_names_inp.is_open()) {
        cout << "\"tnames_systemc.out\" is loading.." << endl;
        string name;
        while (thread_names_inp >> name) thread_names.push_back(name);
        thread_names_inp.close();
    } else {
        cout << "\"tnames_systemc.out\" cannot be loaded." << endl;
    }

    // Initialize the lock
    InitLock(&lock);

    // Obtain  a key for TLS storage.
    tls_key = PIN_CreateThreadDataKey(0);

    INS_AddInstrumentFunction(Instruction, 0);
    IMG_AddInstrumentFunction(Image, 0);
    PIN_AddFiniFunction(Fini, 0);

    PIN_AddThreadStartFunction(ThreadStart, 0);
```

```
    PIN_AddThreadFiniFunction(ThreadFini, 0);

    // Never returns
    PIN_StartProgram();

    return 0;
}
```

APPENDIX B: P2S Example

B.1. blackscholes_syn.c

```
/*
 * Copyright (c) 2011-2013, Etem Deniz <etem.deniz@boun.edu.tr>, Bogazici University
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * (1) Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * (2) Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution.
 *
 * (3) The names of the contributors may not be used to endorse or promote
 * products derived from this software without specific prior written
 * permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
 * IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER
 * OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 */

/*
 * Description: Synthetic benchmark of blackscholes program.
 *              Parallel pattern of blackscholes program is Task Parallel.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
```

```
#include <unistd.h>
#include <systemc.h>

int globAddr1[197];

SC_MODULE(mod) {
    void task2() {
        int workCount; /* work counter */
        int localMemTemp; /* temp read/write data */
        int localMemIt; /* temp read/write data */
        int localAddr1[1]; /* thread local mem */
        int retVal = -1;

        for (localMemIt = 0; localMemIt < 1; localMemIt++) {
            localAddr1[localMemIt] = localMemIt; /* write local mem */
        }

        /*
         * ================================================================
         * ========================= DO WORK-1 ============================
         * Description: code block for computation
         * ================================================================
         */
        for (workCount = 0; workCount < 1013; workCount++) {
            /* do something */
        }

        for (localMemIt = 0; localMemIt < 1; localMemIt++) {
            localMemTemp = localAddr1[localMemIt]; /* read local mem */
            /* use localMemTemp */
        }
    }

    void task3() {
        int workCount; /* work counter */
        int localMemTemp; /* temp read/write data */
        int localMemIt; /* temp read/write data */
        int localAddr1[1]; /* thread local mem */
        int retVal = -1;

        for (localMemIt = 0; localMemIt < 1; localMemIt++) {
            localAddr1[localMemIt] = localMemIt; /* write local mem */
        }

        /*
         * ================================================================
         * ========================= DO WORK-1 ============================
         * Description: code block for computation
```

```
     * ==========================================================
     */
    for (workCount = 0; workCount < 1007; workCount++) {
        /* do something */
    }

    for (localMemIt = 0; localMemIt < 1; localMemIt++) {
        localMemTemp = localAddr1[localMemIt]; /* read local mem */
        /* use localMemTemp */
    }
}

void task4() {
    int workCount; /* work counter */
    int localMemTemp; /* temp read/write data */
    int localMemIt; /* temp read/write data */
    int localAddr1[1]; /* thread local mem */
    int retVal = -1;

    for (localMemIt = 0; localMemIt < 1; localMemIt++) {
        localAddr1[localMemIt] = localMemIt; /* write local mem */
    }

    /*
     * ==========================================================
     * ========================= DO WORK-1 ======================
     * Description: code block for computation
     * ==========================================================
     */
    for (workCount = 0; workCount < 999; workCount++) {
        /* do something */
    }

    for (localMemIt = 0; localMemIt < 1; localMemIt++) {
        localMemTemp = localAddr1[localMemIt]; /* read local mem */
        /* use localMemTemp */
    }
}

void task5() {
    int workCount; /* work counter */
    int localMemTemp; /* temp read/write data */
    int localMemIt; /* temp read/write data */
    int localAddr1[1]; /* thread local mem */
    int retVal = -1;

    for (localMemIt = 0; localMemIt < 1; localMemIt++) {
        localAddr1[localMemIt] = localMemIt; /* write local mem */
```

```
        }

        /*
         * ============================================================
         * ========================= DO WORK-1 ========================
         * Description: code block for computation
         * ============================================================
         */
        for (workCount = 0; workCount < 1016; workCount++) {
            /* do something */
        }

        for (localMemIt = 0; localMemIt < 1; localMemIt++) {
            localMemTemp = localAddr1[localMemIt]; /* read local mem */
            /* use localMemTemp */
        }
    }

    SC_CTOR(mod) {
        SC_THREAD(task2);
        set_stack_size(0x64000);
        SC_THREAD(task3);
        set_stack_size(0x64000);
        SC_THREAD(task4);
        set_stack_size(0x64000);
        SC_THREAD(task5);
        set_stack_size(0x64000);
    }
};

int sc_main(int argc, char **argv) {
    mod m("modname");
    sc_start();
    register unsigned int workCount; /* work counter */
    int globMemTemp;
    int globMemIt;
    int retVal = -1;

    for (globMemIt = 0; globMemIt < 197; globMemIt++) {
        globMemTemp = globAddr1[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    /*
     * ============================================================
     * ========================= DO WORK-1 ========================
     * Description: code block for computation
     * ============================================================
```

```
     */
    for (workCount = 0; workCount < 1024; workCount++) {
        /* do something */
    }

    /*
     * ============================================================
     * ========================= DO WORK-2 ========================
     * Description: code block for CMR increment
     * ============================================================
     */
    register int wi;
    int *wa, wsize = (1024 * 1024) * 4;
    wa = (int *) malloc(wsize * sizeof(int));
    for (workCount = 0; workCount < 1159; workCount++) {
        /* do something */
        wa[rand() % wsize] = 0;
    }
    free(wa);

    /*
     * ============================================================
     * ========================= DO WORK-3 ========================
     * Description: code block for BMR decrement
     * ============================================================
     */
    int b1 = 288, b2 = 3, bres = 0;
    for (workCount = 0; workCount < 449740; workCount++) {
        /* do something */
        if (workCount >= 0) {
            bres = b1 + b2 + (b1 / b2);
            bres = b1 + b2 + (b1 / b2);
        }
    }

    return 0;
}
```

APPENDIX C: S2P Example

C.1. at_1_phase_syn.cpp

```
/*
 * Copyright (c) 2011-2013, Etem Deniz <etem.deniz@boun.edu.tr>, Bogazici University
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * (1) Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * (2) Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution.
 *
 * (3) The names of the contributors may not be used to endorse or promote
 * products derived from this software without specific prior written
 * permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
 * IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER
 * OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 */

/*
 * Description: Synthetic benchmark of at_1_phase program.
 *              Parallel pattern of at_1_phase program is Recursive Data.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
```

```
#include <unistd.h>
#include <semaphore.h>
#include <errno.h>
#include <fcntl.h>

typedef struct {
    unsigned int tid;
} threadData;

#define WRONG wrong(td->tid, status, __FILE__, __LINE__);
void wrong(unsigned int tid, char *status, char *file, unsigned line) {
    fprintf(stderr, "WRONG: tid=%d, status=%s, file=%s, line=%u\n",
            tid, status, file, line);
    fflush(stdout);
    exit(1);
}

#define POSIX_WRONG posixWrong(td->tid, retVal, __FILE__, __LINE__);
void posixWrong(int tid, int retVal, char *file, unsigned line) {
    fprintf(stderr, "WRONG: tid=%d, retval=%d, errno=%d, file=%s, line=%u\n",
            tid, retVal, errno, file, line);
    fflush(stdout);
    exit(1);
}

int wa[256][256];
int globAddr2[1];
int globAddr3[5001];

/* Task signatures */
void *task0(void *param);
void *task1(void *param);
void *task2(void *param);
void *task3(void *param);
void *task4(void *param);
void *task5(void *param);
void *task6(void *param);
void *task7(void *param);

void *task0(void *param) {
    threadData* td = (threadData*) param;
    int workCount; /* work counter */
    int globMemTemp;
    int globMemIt;
    int retVal = -1;

    for (globMemIt = 0; globMemIt < 442; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
```

```
        /* use globMemTemp */
    }

    for (globMemIt = 442; globMemIt < 503; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    for (globMemIt = 503; globMemIt < 551; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    for (globMemIt = 551; globMemIt < 573; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    /*
     * ================================================================
     * ========================= DO WORK-1 ============================
     * Description: code block for computation
     * ================================================================
     */
    for (workCount = 0; workCount < 342; workCount++) {
        /* do something */
    }

    for (globMemIt = 1059; globMemIt < 1145; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    for (globMemIt = 3332; globMemIt < 3356; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    for (globMemIt = 4551; globMemIt < 4667; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    return NULL;
}

void *task1(void *param) {
    threadData* td = (threadData*) param;
```

```
int workCount; /* work counter */
int globMemTemp;
int globMemIt;
int retVal = -1;

for (globMemIt = 573; globMemIt < 1059; globMemIt++) {
    globMemTemp = globAddr3[globMemIt]; /* read global mem */
    /* use globMemTemp */
}

for (globMemIt = 1059; globMemIt < 1145; globMemIt++) {
    globMemTemp = globAddr3[globMemIt]; /* read global mem */
    /* use globMemTemp */
}

for (globMemIt = 1145; globMemIt < 1195; globMemIt++) {
    globMemTemp = globAddr3[globMemIt]; /* read global mem */
    /* use globMemTemp */
}

for (globMemIt = 1195; globMemIt < 1256; globMemIt++) {
    globMemTemp = globAddr3[globMemIt]; /* read global mem */
    /* use globMemTemp */
}

for (globMemIt = 1256; globMemIt < 1268; globMemIt++) {
    globMemTemp = globAddr3[globMemIt]; /* read global mem */
    /* use globMemTemp */
}

/*
 * ================================================================
 * ========================= DO WORK-1 ============================
 * Description: code block for computation
 * ================================================================
 */
for (workCount = 0; workCount < 425; workCount++) {
    /* do something */
}

for (globMemIt = 442; globMemIt < 503; globMemIt++) {
    globMemTemp = globMemIt; /* assign globMemTemp */
    globAddr3[globMemIt] = globMemTemp; /* write global mem */
}

for (globMemIt = 2340; globMemIt < 2401; globMemIt++) {
    globMemTemp = globMemIt; /* assign globMemTemp */
    globAddr3[globMemIt] = globMemTemp; /* write global mem */
```

```
    }

    for (globMemIt = 3356; globMemIt < 3490; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    for (globMemIt = 4667; globMemIt < 4759; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    return NULL;
}

void *task2(void *param) {
    threadData* td = (threadData*) param;
    int workCount; /* work counter */
    int globMemTemp;
    int globMemIt;
    int retVal = -1;

    for (globMemIt = 1268; globMemIt < 1708; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    for (globMemIt = 1708; globMemIt < 1769; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    for (globMemIt = 1769; globMemIt < 1817; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    for (globMemIt = 1817; globMemIt < 1841; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    /*
     * ============================================================
     * ========================= DO WORK-1 ========================
     * Description: code block for computation
     * ============================================================
     */
```

```
    for (workCount = 0; workCount < 368; workCount++) {
        /* do something */
    }

    for (globMemIt = 2401; globMemIt < 2487; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    for (globMemIt = 3490; globMemIt < 3514; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    for (globMemIt = 4759; globMemIt < 4883; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    return NULL;
}

void *task3(void *param) {
    threadData* td = (threadData*) param;
    int workCount; /* work counter */
    int globMemTemp;
    int globMemIt;
    int retVal = -1;

    for (globMemIt = 1841; globMemIt < 2340; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    for (globMemIt = 2340; globMemIt < 2401; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    for (globMemIt = 2401; globMemIt < 2487; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    for (globMemIt = 2487; globMemIt < 2535; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }
```

```
    for (globMemIt = 2535; globMemIt < 2547; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    /*
     * ================================================================
     * ========================= DO WORK-1 ============================
     * Description: code block for computation
     * ================================================================
     */
    for (workCount = 0; workCount < 462; workCount++) {
        /* do something */
    }

    for (globMemIt = 1145; globMemIt < 1195; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    for (globMemIt = 1708; globMemIt < 1769; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    for (globMemIt = 3514; globMemIt < 3662; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    for (globMemIt = 4883; globMemIt < 4969; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    return NULL;
}

void *task4(void *param) {
    threadData* td = (threadData*) param;
    int workCount; /* work counter */
    int globMemTemp;
    int globMemIt;
    int retVal = -1;

    for (globMemIt = 2547; globMemIt < 3332; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
```

```
    /* use globMemTemp */
}

for (globMemIt = 3332; globMemIt < 3356; globMemIt++) {
    globMemTemp = globAddr3[globMemIt]; /* read global mem */
    /* use globMemTemp */
}

for (globMemIt = 3356; globMemIt < 3490; globMemIt++) {
    globMemTemp = globAddr3[globMemIt]; /* read global mem */
    /* use globMemTemp */
}

for (globMemIt = 3490; globMemIt < 3514; globMemIt++) {
    globMemTemp = globAddr3[globMemIt]; /* read global mem */
    /* use globMemTemp */
}

for (globMemIt = 3514; globMemIt < 3662; globMemIt++) {
    globMemTemp = globAddr3[globMemIt]; /* read global mem */
    /* use globMemTemp */
}

for (globMemIt = 3662; globMemIt < 3846; globMemIt++) {
    globMemTemp = globAddr3[globMemIt]; /* read global mem */
    /* use globMemTemp */
}

/*
 * ============================================================
 * ========================= DO WORK-1 ========================
 * Description: code block for computation
 * ============================================================
 */
for (workCount = 0; workCount < 465; workCount++) {
    /* do something */
}

for (globMemIt = 503; globMemIt < 551; globMemIt++) {
    globMemTemp = globMemIt; /* assign globMemTemp */
    globAddr3[globMemIt] = globMemTemp; /* write global mem */
}

for (globMemIt = 1195; globMemIt < 1256; globMemIt++) {
    globMemTemp = globMemIt; /* assign globMemTemp */
    globAddr3[globMemIt] = globMemTemp; /* write global mem */
}
```

```
    for (globMemIt = 1769; globMemIt < 1817; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    for (globMemIt = 2487; globMemIt < 2535; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    for (globMemIt = 4969; globMemIt < 4983; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    return NULL;
}

void *task5(void *param) {
    threadData* td = (threadData*) param;
    int workCount; /* work counter */
    int globMemTemp;
    int globMemIt;
    int retVal = -1;

    for (globMemIt = 3846; globMemIt < 4551; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    for (globMemIt = 4551; globMemIt < 4667; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    for (globMemIt = 4667; globMemIt < 4759; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    for (globMemIt = 4759; globMemIt < 4883; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    for (globMemIt = 4883; globMemIt < 4969; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
```

```
    }

    for (globMemIt = 4969; globMemIt < 4983; globMemIt++) {
        globMemTemp = globAddr3[globMemIt]; /* read global mem */
        /* use globMemTemp */
    }

    /*
     * ============================================================
     * ========================= DO WORK-1 ========================
     * Description: code block for computation
     * ============================================================
     */
    for (workCount = 0; workCount < 469; workCount++) {
        /* do something */
    }

    for (globMemIt = 551; globMemIt < 573; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    for (globMemIt = 1256; globMemIt < 1268; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    for (globMemIt = 1817; globMemIt < 1841; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    for (globMemIt = 2535; globMemIt < 2547; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    for (globMemIt = 3662; globMemIt < 3846; globMemIt++) {
        globMemTemp = globMemIt; /* assign globMemTemp */
        globAddr3[globMemIt] = globMemTemp; /* write global mem */
    }

    return NULL;
}

void *task6(void *param) {
    threadData* td = (threadData*) param;
    int workCount; /* work counter */
```

```
    int retVal = -1;

    /*
     * ============================================================
     * ========================= DO WORK-1 ========================
     * Description: code block for computation
     * ============================================================
     */
    for (workCount = 0; workCount < 471; workCount++) {
        /* do something */
    }

    return NULL;
}

void *task7(void *param) {
    threadData* td = (threadData*) param;
    int workCount; /* work counter */
    int retVal = -1;

    /*
     * ============================================================
     * ========================= DO WORK-1 ========================
     * Description: code block for computation
     * ============================================================
     */
    for (workCount = 0; workCount < 479; workCount++) {
        /* do something */
    }

    return NULL;
}

int main(int argc, char **argv) {
    threadData maintData = { 1 };
    threadData *td = &maintData;
    threadData tData[8];
    pthread_t threads[8];
    int rc;
    int t;
    int n;
    register unsigned int workCount; /* work counter */
    int globMemTemp;
    int globMemIt;
    int retVal = -1;

    tData[0].tid = 2;
    tData[1].tid = 3;
```

```
tData[2].tid = 4;
tData[3].tid = 5;
tData[4].tid = 6;
tData[5].tid = 7;
tData[6].tid = 8;
tData[7].tid = 9;

for (globMemIt = 0; globMemIt < 1; globMemIt++) {
    globMemTemp = globAddr2[globMemIt]; /* read global mem */
    /* use globMemTemp */
}

for (globMemIt = 0; globMemIt < 5001; globMemIt++) {
    globMemTemp = globMemIt; /* assign globMemTemp */
    globAddr3[globMemIt] = globMemTemp; /* write global mem */
}

/*
 * ================================================================
 * ========================= DO WORK-1 ============================
 * Description: code block for computation
 * ================================================================
 */
for (workCount = 0; workCount < 1024; workCount++) {
    /* do something */
}

/* Create and run all the threads of phase 1 */
t = 0;

rc = pthread_create(&threads[t], NULL, task0, (void *) &tData[t]);
if (rc) {
    fprintf(stderr,"ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
}
t++;

rc = pthread_create(&threads[t], NULL, task1, (void *) &tData[t]);
if (rc) {
    fprintf(stderr,"ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
}
t++;

rc = pthread_create(&threads[t], NULL, task2, (void *) &tData[t]);
if (rc) {
    fprintf(stderr,"ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
```

```
}
t++;

rc = pthread_create(&threads[t], NULL, task3, (void *) &tData[t]);
if (rc) {
    fprintf(stderr,"ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
}
t++;

rc = pthread_create(&threads[t], NULL, task4, (void *) &tData[t]);
if (rc) {
    fprintf(stderr,"ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
}
t++;

rc = pthread_create(&threads[t], NULL, task5, (void *) &tData[t]);
if (rc) {
    fprintf(stderr,"ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
}
t++;

rc = pthread_create(&threads[t], NULL, task6, (void *) &tData[t]);
if (rc) {
    fprintf(stderr,"ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
}
t++;

rc = pthread_create(&threads[t], NULL, task7, (void *) &tData[t]);
if (rc) {
    fprintf(stderr,"ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
}
t++;

/* Wait for all threads of phase 1*/
for (t = 0; t < 8; t++) {
    pthread_join(threads[t], NULL);
}

for (globMemIt = 0; globMemIt < 5001; globMemIt++) {
    globMemTemp = globAddr3[globMemIt]; /* read global mem */
    /* use globMemTemp */
}
```

```
for (globMemIt = 0; globMemIt < 442; globMemIt++) {
    globMemTemp = globMemIt; /* assign globMemTemp */
    globAddr3[globMemIt] = globMemTemp; /* write global mem */
}

for (globMemIt = 573; globMemIt < 1059; globMemIt++) {
    globMemTemp = globMemIt; /* assign globMemTemp */
    globAddr3[globMemIt] = globMemTemp; /* write global mem */
}

for (globMemIt = 1268; globMemIt < 1708; globMemIt++) {
    globMemTemp = globMemIt; /* assign globMemTemp */
    globAddr3[globMemIt] = globMemTemp; /* write global mem */
}

for (globMemIt = 1841; globMemIt < 2340; globMemIt++) {
    globMemTemp = globMemIt; /* assign globMemTemp */
    globAddr3[globMemIt] = globMemTemp; /* write global mem */
}

for (globMemIt = 2547; globMemIt < 3332; globMemIt++) {
    globMemTemp = globMemIt; /* assign globMemTemp */
    globAddr3[globMemIt] = globMemTemp; /* write global mem */
}

for (globMemIt = 3846; globMemIt < 4551; globMemIt++) {
    globMemTemp = globMemIt; /* assign globMemTemp */
    globAddr3[globMemIt] = globMemTemp; /* write global mem */
}

/*
 * ============================================================
 * ========================= DO WORK-2 ========================
 * Description: code block for IPC decrement
 * ============================================================
 */
double d1 = 288, d2 = 3, dres = 0;
for (workCount = 0; workCount < 585255; workCount++) {
    /* do something */
    dres = d1 / d2;
    dres = d1 / d2;
    dres = d1 / d2;
    dres = d1 / d2;
    dres = d1 / d2;
    dres = d1 / d2;
    dres = d1 / d2;
    dres = d1 / d2;
    dres = d1 / d2;
```

```
    dres = d1 / d2;
    dres = d1 / d2;
    dres = d1 / d2;
```

```
     */
    int i, j;
    for (workCount = 0; workCount < 100; workCount++) {
        /* do something */
        for (i = 0; i < 256; i += 1) {
            for (j = 0; j < 256; j += 1) {
                wa[i][j] = 2 * wa[i][j];
            }
        }
    }

    return 0;
}
```

REFERENCES

1. Deniz, E., A. Sen, J. Holt and B. Kahne, "Using Software Architectural Patterns for Synthetic Embedded Multicore Benchmark Development", *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 89–99, IEEE, 2012.

2. Mattson, T. G., B. A. Sanders and B. L. Massingill, *Patterns for Parallel Programming*, Pearson Education, 2004.

3. Poovey, J. A., B. P. Railing and T. M. Conte, "Parallel Pattern Detection for Architectural Improvements", *Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar)*, p. 12, USENIX Association, 2011.

4. Aynsley, J., *OSCI TLM-2.0 Language Reference Manual*, Open SystemC Initiative (OSCI), 2009.

5. Bienia, C., *Benchmarking Modern Multiprocessors*, Ph.D. Thesis, Princeton University, 2011.

6. Open SystemC Initiative (OSCI), *IEEE Standard SystemC Language Reference Manual*, 2006.

7. Ganesan, K. and L. K. John, "Automatic Generation of Miniaturized Synthetic Proxies for Target Applications to Efficiently Design Multicore Processors", *IEEE Transactions on Computers (TC)*, Vol. 63, No. 4, pp. 833–846, 2014.

8. Kreku, J., K. Tiensyrjä and G. Vanmeerbeeck, "Automatic Workload Generation for System-Level Exploration Based on Modified GCC Compiler", *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pp. 369–374, European Design and Automation Association, 2010.

9. Joshi, A., L. Eeckhout, R. H. B. Jr. and L. K. John, "Distilling the Essence of

Proprietary Workloads into Miniature Benchmarks", *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 5, No. 2, p. 10, 2008.

10. Coffland, J. E. and A. D. Pimentel, "A Software Framework for Efficient System-level Performance Evaluation of Embedded Systems", *Proceedings of the 2003 ACM Symposium on Applied Computing (SAC)*, pp. 666–671, ACM, 2003.

11. Pimentel, A. D., "The Artemis Workbench for System-Level Performance Evaluation of Embedded Systems", *International Journal of Embedded Systems (IJES)*, Vol. 3, No. 3, pp. 181–196, 2008.

12. Lahiri, K., A. Raghunathan and S. Dey, "System-Level Performance Analysis for Designing On-Chip Communication Architectures", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 20, No. 6, pp. 768–783, 2001.

13. Henia, R., A. Hamann, M. Jersak, R. Racu, K. Richter and R. Ernst, "System Level Performance Analysis - The SymTA/S Approach", *IEE Proceedings Computers and Digital Techniques*, Vol. 152, No. 2, pp. 148–166, 2005.

14. Alameldeen, A. R. and D. A. Wood, "IPC Considered Harmful for Multiprocessor Workloads", *IEEE Micro*, Vol. 26, No. 4, pp. 8–17, 2006.

15. Eyerman, S. and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads", *IEEE Micro*, Vol. 28, No. 3, pp. 42–53, 2008.

16. Künzli, S., F. Poletti, L. Benini and L. Thiele, "Combining Simulation and Formal Methods for System-Level Performance Analysis", *Proceedings of Design, Automation and Test in Europe (DATE)*, Vol. 1, pp. 236–241, IEEE, 2006.

17. Grammatikakis, M. D., A. Papagrigoriou, P. Petrakis and G. Kornaros, "Monitoring-Aware Virtual Platform Prototype of Heterogeneous NoC-Based Multicore SoCs", *Euromicro Conference on Digital System Design (DSD)*, pp. 497–504,

IEEE, 2013.

18. Beltrame, G., L. Fossati and D. Sciuto, "A Real-Time Application Design Methodology for MPSoCs", *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pp. 767–772, European Design and Automation Association, 2009.

19. Shen, H., P. Gerin and F. Pétrot, "Configurable Heterogeneous MPSOC Architecture Exploration using Abstraction Levels", *The 19th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP)*, pp. 51–57, IEEE, 2008.

20. Chureau, A. and F. Pétrot, "An Intermediate Format for Automatic Generation of MPSoC Virtual Prototypes", *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS)*, pp. 165–172, IEEE, 2008.

21. Posadas, H., J. Ádamez, E. Villar, F. Blasco and F. Escuder, "RTOS Modeling in SystemC for Real-Time Embedded SW Simulation: A POSIX Model", *Design Automation for Embedded Systems*, Vol. 10, No. 4, pp. 209–227, 2005.

22. Fummi, F., M. Loghi, G. Perbellini and M. Poncino, "SystemC Co-Simulation for Core-Based Embedded Systems", *Design Automation for Embedded Systems*, Vol. 11, No. 2-3, pp. 141–166, 2007.

23. Vinco, S., D. Chatterjee, V. Bertacco and F. Fummi, "SAGA: SystemC Acceleration on GPU Architectures", *49th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 115–120, IEEE, 2012.

24. Nanjundappa, M., H. D. Patel, B. A. Jose and S. K. Shukla, "SCGPSIM: A Fast SystemC Simulator on GPUs", *15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 149–154, IEEE, 2010.

25. Accellera Systems Initiative, *Systemc 2.3 (Includes TLM)*, 2012.

26. Melo, A., "The New Linux 'perf' Tools", *17 International Linux System Technology Conference (Linux Kongress)*, pp. 21–24.

27. Luk, C.-K., R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi and K. M. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation", *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, Vol. 40, No. 6, pp. 190–200, 2005.

28. Nethercote, N. and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation", *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, Vol. 42, No. 6, pp. 89–100, 2007.

29. Bruening, D. L., *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*, Ph.D. Thesis, Massachusetts Institute of Technology, 2004.

30. Bird, S., A. Phansalkar, L. K. John, A. Mericas and R. Indukuru, "Performance Characterization of SPEC CPU Benchmarks on Intels Core Microarchitecture Based Processor", *SPEC Benchmark Workshop*, 2007.

31. Ganesan, K., L. K. John, V. Salapura and J. C. Sexton, "A Performance Counter Based Workload Characterization on Blue Gene/P", *37th International Conference on Parallel Processing (ICPP)*, pp. 330–337, IEEE, 2008.

32. Hoste, K. and L. Eeckhout, "Microarchitecture-Independent Workload Characterization", *IEEE Micro*, Vol. 27, No. 3, pp. 63–72, 2007.

33. *Getting Started with TLM-2.0*, `http://www.doulos.com/knowhow/systemc/tlm2/`, [Accessed June 2012].

34. Broeders, H. and R. van Leuken, "Extracting Behavior and Dynamically Generated Hierarchy from SystemC Models", *Proceedings of the 48th Design Automation Conference (DAC)*, pp. 357–362, ACM, 2011.

35. Cimatti, A., I. Narasamdya and M. Roveri, "Software Model Checking SystemC",

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 32, No. 5, pp. 774–787, 2013.

36. Chou, C.-N., Y.-S. Ho, C. Hsieh and C.-Y. R. Huang, "Symbolic Model Checking on SystemC Designs", *49th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 327–333, IEEE, 2012.

37. Eeckhout, L., H. Vandierendonck and K. D. Bosschere, "Quantifying the Impact of Input Data Sets on Program Behavior and its Applications", *Journal of Instruction-Level Parallelism (JILP)*, Vol. 5, No. 1, pp. 1–33, 2003.

38. Greaves, D. J. and M. Yasin, "TLM POWER3: Power Estimation Methodology for SystemC TLM 2.0", *Models, Methods, and Tools for Complex Chip Design*, pp. 53–68, 2014.

39. Grammatikakis, M. D., S. Politis, J.-P. Schoellkopf and C. Papadas, "System-Level Power Estimation Methodology using Cycle- and Bit-Accurate TLM", *Design, Automation and Test in Europe (DATE)*, pp. 1125–1126, IEEE, 2011.

40. Streubuhr, M., R. Rosales, R. Hasholzner, C. Haubelt and J. Teich, "ESL Power and Performance Estimation for Heterogeneous MPSoCs using SystemC", *Forum on Specification and Design Languages (FDL)*, pp. 1–8, IEEE, 2011.